DOINA LOGOFĂTU

Algoritmi fundamentali în C++

Aplicaţii

Doina Logofătu (n. 1974) este absolventă a Facultății de Informatică, Universitatea „Al.I. Cuza" din Iași. În 1999 a absolvit cursurile de studii aprofundate (masterat), cu lucrarea de disertație *Programarea distribuită în Java RMI, o analiză comparativă*. A predat la Liceul de Informatică „Grigore C. Moisil" din Iași, a susținut laboratoare de C/C++ la Facultatea de Informatică din Iași, a lucrat ca programator la firme de soft din Iași și München. În prezent, lucrează ca programator în München. Este coautoare a manualului de informatică pentru clasa a XI-a, profilul matematică-informatică (Petrion, 2001), autoare a lucrărilor *C++. Probleme rezolvate și algoritmi* (Polirom, 2001), *Bazele programării în C. Aplicații* (Polirom, 2006) și a unor articole publicate în revista *GInfo*. A mai publicat în Germania volumul *Algorithmen und Problemlösungen mit C++* (Vieweg, 2006) și lucrări la conferințe internaționale.

© 2007 by Editura POLIROM

www.polirom.ro

Editura POLIROM
Iași, B-dul Carol I nr. 4, P.O. BOX 266, 700506
București, B-dul I.C. Brătianu nr. 6, et. 7, ap. 33, O.P. 37; P.O. BOX 1-728, 030174

Descrierea CIP a Bibliotecii Naționale a României:

LOGOFĂTU, DOINA

Algoritmi fundamentali în C++ / Doina Logofătu. – Iași:
Polirom, 2007

ISBN 978-973-46-0093-9

004.43 C++

Printed in ROMANIA

Doina Logofătu

Algoritmi fundamentali în C++

Aplicații

POLIROM
2007

Surorii mele Monica

Cuprins

Cuvânt înainte 9

Capitolul 1. Algoritmi - elemente definitorii 11
Geneza cuvântului algoritm 13
Definiţii alternative 13
Exemple 14
De la problemă spre soluţie 18
Proprietăţile unui algoritm 21
Algoritmica 21
Modelul de calcul *RAM* 22
Complexitatea algoritmilor 23
Exemplu comparativ de creştere a lui $O(g(n))$ 27
Timpul real necesar unui algoritm (polinomial *vs* exponenţial) 27
Clasificarea problemelor (*P*, *NP*, *NP*-complete, *NP-hard*) 28
Probleme *NP*-complete 29
Problema satisfiabilităţii (*SAT*) 30
Clasa problemelor *NP-hard* 31
Exerciţii şi probleme propuse 31

Capitolul 2. Introducere în POO 35
Concepte de bază 37
Proprietăţi ale programării orientate obiect 41
Crearea şi distrugerea obiectelor 43
Elemente statice ale obiectelor 44
Avantajele utilizării programării orientate obiect 45
Probleme propuse 45

Capitolul 3. Introducere în *STL* 47
Istoric, concepte de bază 49
Programarea generică 49
Containere 51
Iteratori 55
Algoritmi 58
Functori 61
Încă şapte probleme rezolvate 63
Întrebări, exerciţii, probleme propuse 72

Capitolul 4. Cutii ascunse 75
Descrierea problemei 77
Analiza problemei şi proiectarea soluţiei 78
Algoritmul 79
Programul 81
Analiza programului 84
Trei mici trucuri de programare 85
Probleme propuse 86
Observaţii 86

Capitolul 5. Sume de puteri 89
Descrierea problemei 91
Analiza problemei. Modelarea algebrică 92
De la formula recursivă spre algoritm 93
Algoritmul 96
Programul 98
Probleme propuse 101

Capitolul 6. *Greedy* 103
Fundamente 105
Problema 1. Problema fracţionară a rucsacului 106
Problema 2. Colorarea hărţii 110
Problema 3. Calul pe tabla de şah 112
Problema 4. Arborele parţial de cost minim (algoritmul lui Kruskal) 115
Problema 5. Codificarea Huffman 119

Capitolul 7. Problema ordonării datelor 127
Descrierea problemei 129
Domeniul problemei. Definiţii 129
POD şi *PODI* sunt *NP*-complete 134
Algoritmi pentru *POD* şi *PODI* 135
Algoritmii *Random* (*RAN*) 136
Algoritmii *Exact* (*EX*) 137
Algoritmii *Greedy Min* (*GM*) 138
Algoritmii *Greedy Min Simplified* (*GMS*) 138
Algoritmii de mărginire inferioară (*LB*) 138
Detalii de implementare 140
Program (*STL*, parametri în linia de comandă) 143
Interpretarea rezultatelor 150
Probleme propuse 153

Capitolul 8. Recursivitatea 155
Inducţia matematică 157
Recursivitatea. Fundamente 163
Problema 1. Suma cifrelor şi oglinditul unui număr natural 164
Problema 2. Numărul 4 166

Capitolul 4. Cutii ascunse 75
Descrierea problemei 77
Analiza problemei şi proiectarea soluţiei 78
Algoritmul 79
Programul 81
Analiza programului 84
Trei mici trucuri de programare 85
Probleme propuse 86
Observaţii 86

Capitolul 5. Sume de puteri 89
Descrierea problemei 91
Analiza problemei. Modelarea algebrică 92
De la formula recursivă spre algoritm 93
Algoritmul 96
Programul 98
Probleme propuse 101

Capitolul 6. *Greedy* 103
Fundamente 105
Problema 1. Problema fracţionară a rucsacului 106
Problema 2. Colorarea hărţii 110
Problema 3. Calul pe tabla de şah 112
Problema 4. Arborele parţial de cost minim (algoritmul lui Kruskal) 115
Problema 5. Codificarea Huffman 119

Capitolul 7. Problema ordonării datelor 127
Descrierea problemei 129
Domeniul problemei. Definiţii 129
POD şi *PODI* sunt *NP*-complete 134
Algoritmi pentru *POD* şi *PODI* 135
Algoritmii *Random* (*RAN*) 136
Algoritmii *Exact* (*EX*) 137
Algoritmii *Greedy Min* (*GM*) 138
Algoritmii *Greedy Min Simplified* (*GMS*) 138
Algoritmii de mărginire inferioară (*LB*) 138
Detalii de implementare 140
Program (*STL*, parametri în linia de comandă) 143
Interpretarea rezultatelor 150
Probleme propuse 153

Capitolul 8. Recursivitatea 155
Inducţia matematică 157
Recursivitatea. Fundamente 163
Problema 1. Suma cifrelor şi oglinditul unui număr natural 164
Problema 2. Numărul 4 166

Problema 3. *Big Mod* 167
Problema 4. Tortul (recursivitate liniară) 169
Problema 5. Funcţia Ackermann (recursivitate imbricată – *compound recursion*) 171
Problema 6. Transformare recursivă în altă bază (din baza 10 în baza *p*) 173
Problema 7. Suma a două rădăcini (recursivitate ramificată) 175
Problema 8.Funcţia lui Collatz (recursivitate nemonotonă) 176
Problema 9. Pătrate şi pătrăţele 178
Problema 10. Pătrate (recursivitate directă) 181
Problema 11. Pătrate şi cercuri (recursivitate indirectă) 190
Problema 12. Curba fulgului de zăpadă a lui Koch 194

Capitolul 9. *Divide et Impera* 203
Fundamente 205
Problema 1. Cel mai mare divizor comun 206
Problema 2. Turnurile din Hanoi 207
Problema 3. Integrală cu regula trapezului 209
Problema 4. *QuickSort* (sortarea rapidă) 211
Problema 5. *MergeSort* (sortarea prin interclasare) 213
Problema 6. Arbori *Quad* (*Quad-Trees*) 215
Problema 7. Transformata *Fourier* discretă (*DFT*) 220

Capitolul 10. *Backtracking* 225
Problema 1. Problema celor *n* regine 227
Observaţii generale cu privire la metoda *Backtracking* 232
Problema 2. Problema celor *n* ture pe tabla de şah 234
Problema 3. Problema turelor pe primele *m* linii ale unei table de şah 235
Problema 4. Problema turelor în scară pe primele *m* linii ale unei table de şah ... 236
Problema 5. Tabăra prieteniei 237
Problema 6. Partiţiile unui număr natural 238
Problema 7. Referate la geografie 241
Problema 8. Toate drumurile calului pe tabla de şah 244
Problema 9. Problema fotografiei 246
Problema 10. Rostogolirea mingii 248
Problema 11. Portocalosport 250
Problema 12. Sudoku 257
Problema 13. Căsuţa lui Moş Crăciun 263
Problema 14. Compactarea şabloanelor de test 265
Încă 12 probleme propuse 273

Capitolul 11. Programarea dinamică 283
Fundamente şi proprietăţi ale metodei 285
Problema 1. Numărarea iepuraşilor 289
Problema 2. Subşir crescător maximal 293
Problema 3. Cel mai lung subşir comun (*LCS*) 296
Problema 4. Triunghi de numere 300
Problema 5. Domino 303

Problema 6. Împărţirea cadourilor 306
Problema 7. Sume asemănătoare 309
Problema 8. Scoţieni la Oktoberfest 314
Problema 9. Calul pe tabla de şah 320
Problema 10. Fluctuaţii la bursă 326
Problema 11. Sume de produse 329
Problema 12. Triangularizare minimală a unui poligon convex 333
Problema 13. Înmulţirea unui şir de matrice 339
Problema 14. *Edit-Distance* 344

Bibliografie 353
Glosar 356

Cuvânt înainte

Această carte se bazează preponderent pe o parte a volumului *Algorithmen und Problemlösungen mit C++*, care a apărut în Germania (octombrie 2006). Cartea de față cuprinde noi probleme rezolvate (ex. Codificarea Huffman, Sudoku, *LCS*) în cadrul capitolelor corespondente, dar şi noi capitole (ex. „Introducere în *STL*", „Problema ordonării datelor"). O serie de programe au fost refăcute şi extinse, au fost de asemenea adăugate noi figuri, imagini şi tabele. Cartea cuprinde experienţe, observaţii şi cunoştinţe pe care le-am adunat în decursul a peste 10 ani. În studenţie, mi-am dorit cărţi care să uşureze accesul spre teorie şi să conţină multe exemple practice. Ca profesoară, mi-am dorit cărţi care să conţină probleme atractive, care să mărească interesul şi curiozitatea elevilor pentru informatică. Ca programator, mi-am dorit cărţi în care să găsesc soluţii la probleme specifice. Câteodată aş lua o carte în concediu, din care să citesc mici povestiri şi în acelaşi timp să repet concepte teoretice, fără să trebuiască să mă pierd în multe şi complicate formalizări. Acum, când încerc să găsesc motivele pentru care e nevoie de această carte, mă gândesc la toate aceste lucruri şi cred că, vrând-nevrând, am încercat să le cuprind pe toate sub acelaşi acoperiş.

Algoritmi fundamentali în C++ cuprinde în jur de 70 de probleme complet analizate şi rezolvate în C++, 250 de probleme propuse şi 130 de figuri şi imagini. Primele trei capitole conţin aspecte generale, teoretice despre algoritmi, programarea orientată obiect şi *Standard Template Library (STL)*. Fiecare capitol începe cu o secţiune de fundamente, care face posibilă o privire de ansamblu asupra trăsăturilor metodei. Pentru fiecare problemă se descrie cum sunt construite fişierele de intrare şi ieşire şi este furnizat un exemplu sugestiv. În acest fel puteţi verifica corectitudinea unor soluţii proprii. Secţiunea *Analiza problemei şi proiectarea soluţiei* prezintă detaliat o soluţie a problemei şi este urmată de *listing*-ul complet al programului C++. Programele sunt compacte şi cuvintele-cheie sunt îngroşate pentru a uşura citirea lor, motiv pentru care comentariile nu se găsesc în cod, ci în casetele gri alăturate. Ele sunt compilate cu Microsoft® Visual C++ 2005 Express Edition, oferit gratuit de Microsoft®, şi încearcă să respecte standardul ANSI C++, astfel încât ar trebui să funcţioneze cu orice compilator de C++. O excepţie o reprezintă ultimele trei programe din capitolul 8, „Recursivitatea", care desenează structuri fractale. Acestea folosesc biblioteca *Active Template Library (ATL)* şi sunt concepute special pentru Microsoft® Windows® (pentru primele două este prezentată şi o implementare Borland C++ 3.1, care funcţionează şi pentru DOS®). Fiecare problemă rezolvată este urmată de

exerciții și probleme propuse, care cer, de exemplu, modificarea sau extinderea programelor expuse sau scrierea altora noi, cu scop recapitulativ sau pentru a îmbunătăți aptitudinile de programare.

Toate problemele se referă la metoda și conceptele expuse la începutul capitolului. Scopul este acela de a învăța teorie prin intermediul aplicării în practică, deci la probleme concrete. Multe dintre acestea sunt clasice, cum ar fi: curba fulgului de zăpadă a lui Koch, turnurile din Hanoi, parcurgerile *DFS* și *BFS* în graf, problema celor *n* regine, colorarea hărții, cel mai lung subșir comun (*LCS*), înmulțirea unui șir de matrice și distanța de editare. Olimpiadele județene și naționale, precum și concursurile de programare *Association for Computing Machinery* (*ACM*), *International Olympiad in Informatics* (*IOI*), *Central-European Olympiad in Informatics* (*CEOI*) au fost surse de inspirație pentru numeroase probleme din carte.

Ca răsplată pentru sârguința dumneavoastră, veți găsi la sfârșitul fiecărui capitol câte o fotografie surpriză, cum ar fi: fan bavarez la Campionatul Mondial de Fotbal (Germania, 2006), arbori în Ottawa, case oglindite în Lübeck sau statuie în Bremen.

Dacă doriți, vă rog să îmi scrieți observațiile și propunerile dumneavoastră la adresa *doinabooks@yahoo.com*, lucru pentru care vă mulțumesc anticipat.

Lectură și învățare plăcute!

Doina Logofătu
München, februarie 2007

Capitolul 1
Algoritmi – elemente definitorii

- Geneza cuvântului *algoritm*
- Definiții alternative
- Patru exemple de algoritmi (algoritmul lui Euclid, „ciurul lui Eratostene", căutarea binară, prepararea prăjiturii Tiramisu)
- De la problemă spre soluție
- Proprietățile unui algoritm
- Algoritmica
- Modelul de calcul *RAM*
- Complexitatea algoritmilor
- Optimalitatea şi reducerea algoritmilor. Exemple
- Exemplu comparativ de creştere a lui $O(g(n))$
- Timpul real necesar unui algoritm (polinomial *vs* exponențial)
- Clasificarea problemelor (*P*, *NP*, *NP*-complete, *NP-hard*)
- Probleme *NP*-complete
- Problema satisfiabilității (*SAT*)
- Clasa problemelor *NP-hard*
- Exerciții şi probleme propuse

Începutul este jumătatea întregului.
Aristotel

Deşi computerele şi aplicaţiile variate corespunzătoare lor sunt un fenomen exploziv ce a devenit predominant în ultimii 20 de ani, o mare parte dintre conceptele ce stau la baza acestor instrumente soft datează de foarte mult timp, chiar din Antichitate, şi ele s-au dezvoltat de-a lungul timpului.

Geneza cuvântului *algoritm*

Cuvântul *algoritm* provine de la numele matematicianului, geografului şi astrologului arab *Abu Ja'far Muhammad ibn Musa Al-Khwarizmi* (780?-850?), care a trăit în Bagdad, capitala Irakului. Cea mai importantă lucrare a sa este *Hisab al-jabr w'al-muqabala*, iar titlul va conduce şi la geneza cuvântului *algebra*, utilizat pentru a defini această ramură a matematicii. El a lucrat la Casa Înţelepciunii din Bagdad, sub patronajul direct al califului Al-Mamun, unde traducea manuscrisele ştiinţifice greceşti şi studia algebra, geometria şi astrologia. Vechii hinduşi, greci, babilonieni, romani şi chinezi foloseau termenul *algoritm* pentru operaţii aritmetice simple.

Definiţii alternative

În linii mari, despre un algoritm putem afirma următoarele:

- Un algoritm este o mulţime de reguli ce se pot aplica în cadrul procesului de construcţie a soluţiei şi pot fi executate fie „de mână", fie de o maşină.
- Un algoritm este o secvenţă de paşi care transformă mulţimea datelor de intrare în datele de ieşire, rezultatele dorite.
- Un algoritm este o secvenţă de operaţii executate cu date ce trebuie organizate în structuri de date.
- Un algoritm este abstractizarea unui program care trebuie executat pe o maşină fizică (model de calcul).
- Un algoritm pentru o problemă dată este o mulţime de instrucţiuni care garantează găsirea unei soluţii corecte pentru orice instanţă a problemei, într-un număr finit de paşi.

Exemple

Algoritmul 1. Algoritmul lui Euclid

Euclid

Nu există dovezi sigure pentru nimic din ceea ce se afirmă astăzi despre Euclid, iar uneori se pune la îndoială chiar şi existenţa acestuia. Se presupune că a trăit în jurul anului 300 î.Ch. şi că a fost un matematician grec, ce a predat la Alexandria, în Egipt. Se pare că el a scris *Elementele*, o lucrare didactică exemplară, care sintetizează cunoştinţele de la acea vreme ale matematicii greceşti. În a şaptea carte se găseşte algoritmul lui Euclid:

Presupunem că a şi b sunt două numere naturale, $a \geq b$, şi că dorim aflarea celui mai mare divizor comun al lor. Considerăm $a_1 = a$ şi $b_1 = b$ şi definim perechile (m_i, r_i), astfel încât $a_i = m_i b_i + r_i$, $0 \leq r_i < b_i$. Mai departe, vom considera $a_{i+1} = b_i$ şi $b_{i+1} = r_i$. Atunci există un număr natural k, astfel încât $r_k = 0$. M mult, dacă $r_k = 0$, $cmmdc(a, b) = r_{k-1}$.

ALGORITM_EUCLID
1. Citeşte $a, b \in \mathbb{N}$, $a \geq b > 0$
2. $a_1 \leftarrow a$, $b_1 \leftarrow b$, $i \leftarrow 1$
3. **While** $(b_i \neq 0)$ **Do**
   - 3.1. $a_{i+1} \leftarrow b_i$
   - 3.2. $b_{i+1} \leftarrow r_i$ $(= a_i \bmod b_i)$
   - 3.3. $i \leftarrow i + 1$

   End_While
4. $ggT(a, b) = r_{i-1}$

END_ALGORITM_EUCLID

Exemple:

a = 294, b = 77	a = 2521, b = 338
294 = 3 x 77 + 63	2521 = 7 x 338 + 155
77 = 1 x 63 + 14	338 = 2 x 155 + 28
63 = 4 x 14 + 7	155 = 5 x 28 + 15
14 = 2 x **7** + 0	28 = 1 x 15 + 13
	15 = 1 x 13 + 2
	13 = 6 x 2 + 1
	2 = 2 x **1** + 0

Algoritmul 2. Ciurul lui Eratostene

Eratostene

Eratostene (?276-194 î.Ch., Cyrene acum Libia, Africa de Nord) a fost primul mare geograf al lumii antice şi este considerat fondatorul geografiei fizice şi matematice. A lucrat mult timp la celebra bibliotecă din Alexandria din Egipt şi a fost, de asemenea, interesat de geometrie şi de numere prime. El a măsurat primul circumferinţa Pământului (care s-a dovedit ulterior corectă) şi a creat o hartă a lumii, pe care a desenat linii paralele. În plus, a mai scris un poem în versuri, *Hermes*, despre fundamentele astronomiei; a sugerat că anii bisecţi se repetă la fiecare 4 ani; a introdus metoda aflării numerelor prime numită „sita lui Eratostene". În memoria sa, există azi un crater pe Lună care îi poartă numele.

Definiţie. Un *număr prim* este un număr natural mai mare decât 1 şi care se divide numai cu 1 şi cu el însuşi. Un număr natural care se divide cu un alt număr decât 1 şi el însuşi se numeşte *număr compus*. „Sita (ciurul) lui Eratostene" este un algoritm care identifică toate numerele prime mai mici sau egale cu un număr natural n dat. Exemplu:

ALGORITM_SITA_ERATOSTENE

1. Se scriu în ordine toate numerele de la 1 la n. (Le vom elimina succesiv pe cele compuse prin marcarea lor. Iniţial, toate numerele sunt nemarcate.)
2. Marchează 1 ca număr special (nu este nici prim, nici compus)
3. $k \leftarrow 1$
4. **Do**
   - 4.1. $m \leftarrow$ primul număr mai mare decât k, dintre cele care nu au fost marcate (primul va fi 2!)
   - 4.2. marchează numerele $2m$, $3m$, $4m$, ... din listă ca fiind compuse (mai întâi se vor marca toţi multiplii lui 2, apoi multiplii lui 3 rămaşi, ...)
   - 4.3. este un număr prim, adaugă-l în lista de numere prime
   - 4.4. $k \leftarrow m$

   While ($k \geq \sqrt{n}$)
5. Adaugă numerele rămase nemarcate în lista numerelor prime
6. Afişează lista numerelor prime până la n

END_ ALGORITM_SITA_ERATOSTENE

eliminarea multiplilor lui 2	eliminarea multiplilor lui 3
(2) 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ 9 ~~10~~ 11 ~~12~~ 13 ~~14~~ 15 ~~16~~ 17 ~~18~~ 19 ~~20~~ 21 ~~22~~ 23 ~~24~~ 25 ~~26~~ 27 ~~28~~ 29 ~~30~~ 31 ~~32~~ 33 ~~34~~ 35 ~~36~~ 37 ~~38~~ 39 ~~40~~ 41 ~~42~~ 43 ~~44~~ 45 ~~46~~ 47 ~~48~~	(2) (3) ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~ 17 ~~18~~ 19 ~~20~~ ~~21~~ ~~22~~ 23 ~~24~~ 25 ~~26~~ ~~27~~ ~~28~~ 29 ~~30~~ 31 ~~32~~ ~~33~~ ~~34~~ 35 ~~36~~ 37 ~~38~~ ~~39~~ ~~40~~ 41 ~~42~~ 43 ~~44~~ ~~45~~ ~~46~~ 47 ~~48~~
eliminarea multiplilor lui 5	**numerele prime mai mici decât 49**
(2) (3) ~~4~~ (5) ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~ 17 ~~18~~ 19 ~~20~~ ~~21~~ ~~22~~ 23 ~~24~~ ~~25~~ ~~26~~ ~~27~~ ~~28~~ 29 ~~30~~ 31 ~~32~~ ~~33~~ ~~34~~ ~~35~~ ~~36~~ 37 ~~38~~ ~~39~~ ~~40~~ 41 ~~42~~ 43 ~~44~~ ~~45~~ ~~46~~ 47 ~~48~~	(2) (3) ~~4~~ (5) ~~6~~ (7) ~~8~~ ~~9~~ ~~10~~ (11) ~~12~~ (13) ~~14~~ ~~15~~ ~~16~~ (17) ~~18~~ (19) ~~20~~ ~~21~~ ~~22~~ (23) ~~24~~ ~~25~~ ~~26~~ ~~27~~ ~~28~~ (29) ~~30~~ (31) ~~32~~ ~~33~~ ~~34~~ ~~35~~ ~~36~~ (37) ~~38~~ ~~39~~ ~~40~~ (41) ~~42~~ (43) ~~44~~ ~~45~~ ~~46~~ (47) ~~48~~

Citiți și implementarea din capitolul 3, problema 10.

Algoritmul 3. Căutarea binară

Presupunând că se dau ca date de intrare un șir finit ordonat crescător de elemente a_1, a_2, ..., a_n și un element x, să se determine prima poziție a lui x în șir, dacă acesta există, altfel, să se returneze valoarea 0.

Informația că șirul de elemente date este deja ordonat ne conduce la determinarea unui algoritm, prin micșorarea succesivă a intervalului de căutat (de exemplu, dacă ultimul element a_n este mai mic decât x, ne oprim deja după o primă comparație și afirmăm că x nu se află în șir). Vom vedea mai târziu de ce acest algorim este în practică mai eficient decât unul în care s-ar efectua căutarea liniară, adică cercetarea tuturor elementelor.

ALGORITM_CĂUTARE_BINARĂ

1. Citeşte $x, a_1, a_2, ..., a_n$
2. $i \leftarrow 1, j \leftarrow n$
3. **While** $(i < j)$ **Do**
   - 3.1. $m \leftarrow (i + j)/2$
   - 3.2. **If** $(x > a_m)$ **Then** $i \leftarrow m + 1$
     - **End_If**
   - 3.3. **Else** $j \leftarrow m$

   End_While
4. **If** $(x = a_i)$ **Then** $poz \leftarrow i$
   End_If
5. **Else** $poz \leftarrow 0$
6. Scrie poz

END_ ALGORITM_CĂUTARE_BINARĂ

Ilustrăm în continuare aplicarea algoritmului pentru şirul ordonat de 17 litere *a c d f g h j l m o p r s u v x z*, în care ne propunem să căutăm litera *j*.

$a_1, \ldots, a_n = a\ c\ d\ f\ g\ h\ j\ l\ m\ o\ p\ r\ s\ u\ v\ x\ z$ $x = j$ $i \leftarrow 1,\ j \leftarrow 17$	
a c d f g h j l m o p r s u v x z *i j* $i = 1, j = 17, m \leftarrow 9$ (3.2. sau 3.3. ?)	*a c d f g h j l m o p r s u v x z* *i j* $i = 1, j = 9$ (pas 3.3), $m \leftarrow 5$ (3.2. sau 3.3. ?)
a c d f g h j l m o p r s u v x z *i j* $i = 5, j = 9$ (pas 3.2), $m \leftarrow 7$ (3.2. sau 3.3. ?)	*a c d f g h j l m o p r s u v x z* *i j* $i = 7, j = 9$ (pas 3.2), $m \leftarrow 8$ (3.2. sau 3.3. ?)
a c d f g h j l m o p r s u v x z *i j* $i = 7, j = 8$ (pas 3.3), $m \leftarrow 7$ (3.2. sau 3.3. ?)	*a c d f g h j l m o p r s u v x z* $(i = j)$ $i = 7, j = 7$ Stop

Pasul 3.2 din algoritm ne indică faptul că dacă numărul *x* căutat se situează la dreapta elementului din mijlocul intervalului cercetat (*m* este mijlocul intervalului [*i*, *j*]), atunci el va trebui căutat în intervalul [*m* + 1, *j*], altfel se va efectua căutarea în intervalul [*i*, *m*], capetele *i*, respectiv *j* modificându-se corespunzător.

Algoritmul 4. Prepararea prăjiturii Tiramisu

Ingrediente: un pachet de biscuiţi (de şampanie sau Löffelbiskuits), 500 g Mascarpone (specialitate de brânză italiană dulce şi moale), 4 ouă, 150 g zahăr, un pahar mic (10 ml) de lichior Amaretto, un pachețel de pudră de gelatină, 3 ml cacao, o cană de cafea îndulcită.

La prima vedere, pare să nu existe nici o legătură între bucătărie şi algoritmică, dar, dacă încercăm să facem o paralelă, vom observa că lucurile nu stau deloc aşa. Rețeta de preparare este, de fapt, un algoritm care, pe baza unor date de intrare (ingredientele), produce datele de ieşire cerute (prăjitura), folosind amestecuri intermediare („structuri de date"). Paşii se execută succesiv şi ordinea de execuție a unor paşi specifici este foarte importantă, exact cum şi o serie de instrucțiuni (paşi) se pot interschimba. De exemplu, nu putem executa pasul 6 înaintea pasului 5, dar putem, în schimb, executa pasul 7 înaintea pasului 5. Observăm cum ingredientele sunt combinate şi transformate pentru a ajunge la îndeplinirea obiectivului final – obținerea desertului. Se poate descrie şi un alt algoritm care va conduce la acelaşi rezultat (de exemplu, dacă biscuiții sunt îmbibați succesiv cu cafea cu ajutorul unei linguriţe, după ce au fost aşezați în tavă). Ca un exercițiu de

imaginație, putem chiar să ne gândim și la construirea unei mașini care să prepare automat Tiramisu.

ALGORITM_PREPARARE_TIRAMISU

1. Start preparare.
2. Răsturnați cafeaua într-o farfurie de supă.
3. Înmuiați biscuiții, ținându-i puțin pe fiecare în farfuria cu cafea (ei trebuie să fie pe jumătate umezi).
4. Așezați biscuiții unul lângă altul în tavă, astfel încât aceștia să „tapeteze" tava.
5. Separați ouăle.
6. Amestecați gălbenușurile, zahărul și lichiorul Amaretto cu mixerul, până se obține o cremă uniformă.
7. Adăugați în crema astfel obținută specialitatea Mascarpone și gelatina și amestecați în continuare.
8. Bateți cu mixerul separat și albușurile până când spuma obținută devine tare (prin întoarcerea bolului, aceasta nu cade).
9. Răsturnați albușurile bătute în crema cu gălbenușuri și amestecați totul cu mixerul.
10. Acoperiți uniform stratul de biscuiți cu o jumătate din cremă. Peste stratul de cremă așezați încă un strat de biscuiți (înmuiați în cafea pe lungime) și peste acest strat distribuiți uniform restul de cremă.
11. Așezați tava în frigider pentru cel puțin trei ore (cel mai bine, peste noapte).
12. Înainte de a se servi, se presară cacao pudră.
13. Sfârșit preparare. Poftă bună!

END_ ALGORITM_PREPARARE_TIRAMISU

De la problemă spre soluție

Problemele din sfera informaticii actuale, indiferent dacă sunt de dimensiune redusă sau adevărate produse gigant, au la bază o serie de idei și caracteristici comune. Comportarea internă în cadrul ciclului de viață al unui program se bazează întotdeauna pe două elmente primordiale: reprezentarea și transformarea.

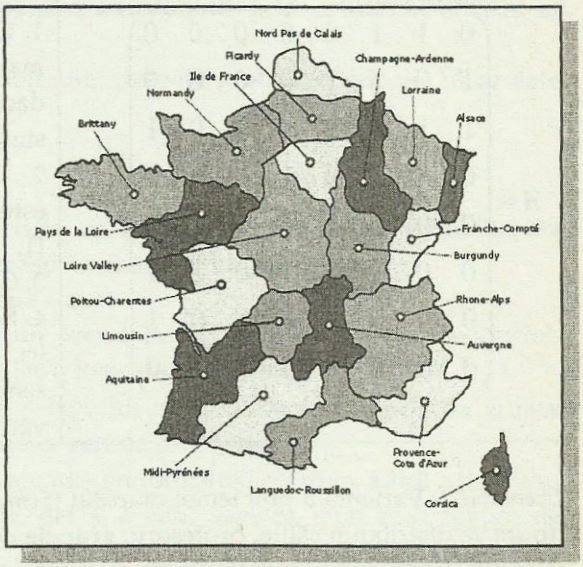
- *Reprezentarea* se referă la o codificare concretă a informației (biți, numere, cuvinte, înregistrări).
- *Transformarea* se referă la modificarea succesivă a stării, prin derularea unor pași (rețetă, program, algoritm), cu scopul de a găsi rezultatele așteptate.

Codificarea simbolică a informației (*reprezentarea*) furnizează o modalitate de comunicare între lumea reală a problemei și lumea „imaginară" a computerului. Se va crea astfel o „mapare", funcție cu ajutorul căreia entitățile reprezentative din cadrul problemei de rezolvat sunt codificate prin valori înțelese de calculator. Această etapă de codificare este foarte importantă, deoarece de ea depind corectitudinea soluției, complexitatea structurilor alese și a algoritmului.

Rezolvarea unei probleme de informatică presupune construirea a trei procese, ce se întrepătrund și se execută secvențial:

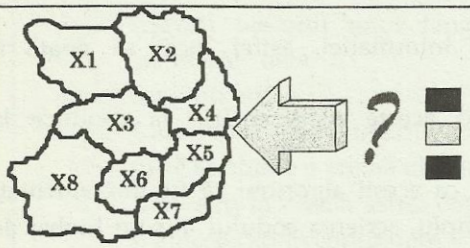
- găsirea unei reprezentări simbolice a informației, astfel încât să poată fi manipulate ulterior;
- formularea algoritmului care, utilizând aceste reprezentări, va conduce la rezultatele dorite;
- construirea cadrului specific ce permite ca aceşti algoritmi să fie implementați într-o manieră corespunzătoare (de exemplu, scrierea codului într-un limbaj de programare).

Exemplul 1. Problema colorării hărții (*Map Colouring Problem*). Dată fiind o hartă cu țări, ne propunem să găsim o modalitate de a colora fiecare țară, astfel încât orice două țări vecine să fie colorate diferit (culorile se vor alege dintr-o paletă de culori dată).



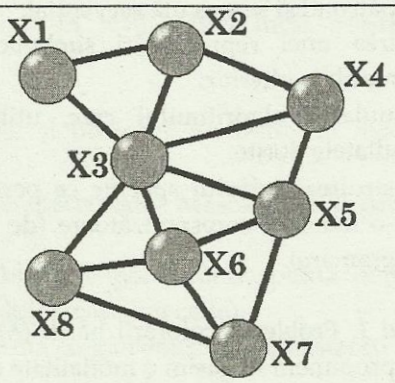
De exemplu, harta Franței va trebui desenată astfel încât oricare două regiuni vecine să fie colorate diferit, folosind culorile roşu, galben, verde şi albastru. Problema apare în forma ei practică şi, pentru a putea fi în general rezolvată, va trebui să o codificăm în aşa fel încât să permitem modelarea ei cu ajutorul unui algoritm. Intuitiv, ne dăm seama că primul pas va fi ataşarea unui număr fiecărei regiuni, de la 1 la numărul acestora n. Mai departe, va trebui să manipulăm şi informațiile de vecinătate, adică am putea construi o matrice pătratică A de dimensiune n x n, în care celula $A(i, j)$ va primi valoarea 1 sau 0, după cum regiunile i şi j sunt – respectiv, nu sunt – vecine. Vom codifica şi culorile cu numerele de la 1 la m, presupunând că dispunem de m culori. În felul acesta, am realizat o

codificare simbolică („mapare") a informației reale, utilizând instrumente matematice, prelucrabile cu ajutorul computerului. Soluția ar trebui să fie un vector de „culori" (numere) $S = \{c_1, c_2, \ldots, c_n\}$, $c_i \in \{1, \ldots, m\}$, cu proprietatea că, dacă $A(i, j) = 1$, atunci $c_i \neq c_j$. Odată stabilită această reprezentare simbolică a informației reale, va trebui să trecem la elaborarea unui algorim care, pe baza matricei A şi a numărului de culori m, să ne furnizeze o soluție S. De fapt, reprezentarea problemei este un graf neorientat cu matricea de adiacență A, în care va trebui să colorăm vârfurile, astfel încât oricare două vârfuri vecine să fie colorate diferit.



8 regiuni, 3 culori

Se cere determinarea unei colorări a regiunilor utilizând cele 3 culori, astfel încât oricare două regiuni vecine să fie colorate diferit.



$$A = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

1. Construirea grafului ataşat hărții şi a matricei de adicență în care $A(i, j) = 1$ dacă şi numai dacă $i \neq j$ şi regiunile i, j sunt vecine.
2. Observăm că matricea de adicență este simetrică: $A(i, j) = A(j, i)$, $\forall i, j \in \{1, \ldots, n\}$.
3. Culorile posibile vor fi $c_1, \ldots, c_m$.
4. Soluția va trebui să fie de forma $S = \{c_1, c_2, \ldots, c_n\}$, $c_i \in \{1, \ldots, m\}$, cu semnificația că c_i este culoarea ataşată vârfului i al grafului.

Exemplul 2. Variantă a problemei orarului (*Schedule Problem*). Concret, presupunem că dorim să distribuim sălile, profesorii, grupele de studenți pentru a realiza orarul unei facultăți. Codificarea informației practice este mult mai dificilă în acest caz decât cea din exemplul anterior, această „mapare" nu se mai produce atât de natural şi accesibil. Va trebui să codificăm cu numere de la 1 la n_p profesorii, numere de la 1 la n_g toate

grupele de studenți (plus fiecare student și apartenența sa la grupe), la fel locațiile (săli, amfiteatre, laboratoare etc.), cursurile posibile. Vor trebui stabilite diverse funcții (caracteristici) care trebuie să fie satisfăcute de o eventuală soluție (fiecare curs are o durată și se poate desfășura doar în anumite săli, o grupă are doar anumite cursuri depinzând de an și specializare, un profesor predă doar anumite cursuri cu anumite grupe etc.) și care trebuie, de asemenea, accesibil codificate. Vor trebui definite și o serie de restricții, astfel încât o posibilă soluție să poată fi verificată matematic (un profesor sau un student nu poate fi în două locuri simultan, într-o anumită sală nu se pot suprapune activități diferite etc.). Numai când are loc definirea în termeni matematici a acestor entități și caracteristici, iar forma soluției este complet realizată, se poate trece la elaborarea unui algoritm pentru a găsi posibilele soluții.

Proprietățile unui algoritm

- Trebuie să posede date de intrare (*input data*).
- Trebuie să furnizeze date de ieșire (*output data*).
- *Determinismul* (la executarea oricărui pas, trebuie să cunoaștem succesorul acestuia).
- *Corectitudinea* datelor de ieșire (a rezultatelor) în raport cu datele de intrare.
- *Finitudinea* (pentru orice set de date de intrare posibile ale problemei, soluția este furnizată într-un număr finit de pași).
- *Eficiența* (furnizarea soluției trebuie să fie realizabilă prin consumul eficient al resurselor timp, spațiu etc.).
- *Generalitatea* (algoritmul este aplicabil unei clase de probleme, cele ale căror date de intrare satisfac anumite condiții).

Algoritmica

Algoritmica este ramura informaticii care se ocupă cu proiectarea și analizarea algoritmilor destinați spre a fi implementați cu ajutorul computerului.

1. Prin *proiectare* (*design*) înțelegem două etape:
   - *descrierea* algoritmului utilizând un pseudolimbaj (schemă logică, pseudocod, descriere cuprinzătoare sugestivă pe o foaie de hârtie etc.);
   - *demonstrarea* corectitudinii, adică a faptului că, indiferent de datele de intrare posibile introduse, rezultatul satisface cerința problemei.
2. Analiza algoritmului se referă la evaluarea performanței acestuia, adică a timpului necesar pentru a determina soluția, dar și a altor aspecte, cum ar fi spațiul de memorie folosit, optimalitatea.

Începutul trebuie să fie alegerea sau definirea unui model de calcul, adică a unui formalism matematic ce descrie interacțiunea dintre componentele sistemului și furnizează abstractizări utile pentru concepte precum *timp* sau *concurență*. De obicei,

modelul folosit este *RAM* (*Random Access Machine*), dar există şi alte modele de calcul, precum: *PRAM*, *Maşini Turing*, automate finite, circuite booleene, automate celulare etc. Odată modelul definit, algoritmul poate fi descris folosind un limbaj simplu, asemănător ca sintaxă; de exemplu, Basic, Pascal, C, C++ , Java etc.

Modelul de calcul *RAM*

Proiectarea unui algoritm independent de maşină depinde de un computer ipotetic numit *Random Access Machine* (*RAM*). Potrivit acestui model, computerul ipotetic satisface:

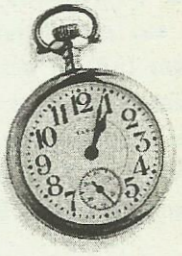
- Fiecare operație „simplă" (+, –, *, /, %, =, *if/dacă*, *call/apelează*) consumă o unitate de timp.
- Ciclurile şi metodele (funcții, proceduri) nu sunt considerate operații simple. Acestea sunt compuse din mai multe operații de un singur pas. Nu ar avea sens să atribuim metodei *sortare*, de exemplu, o unitate de timp, când sortarea unui milion de elemente va consuma mult mai mult timp decât sortarea a 10 elemente. Timpul necesar pentru a executa un ciclu sau o metodă (subprogram) depinde de numărul de iterații, respectiv de natura metodei.
- Fiecare acces la memorie se execută într-o unitate de timp şi avem la dispoziție atâta memorie câtă este necesară. Modelul *RAM* nu face diferența dacă anumite date sunt pe disc sau în *cache* (memoria volatilă), ceea ce simplifică analiza algoritmului.

Conform cu modelul *RAM*, se vor calcula numărul de paşi (unități de timp) necesari pentru instanța unei probleme. Presupunând că *RAM* execută un anumit număr de paşi pe secundă, se poate uşor determina timpul real necesar. Una dintre nemulțumiri ar putea fi faptul că modelul este mult prea simplu, că unele presupuneri sunt prea generale pentru a fi aplicate în practică. De exemplu, pentru înmulțirea a două numere, este nevoie, de obicei, de mai mult timp decât pentru adunarea lor, ceea ce contrazice prima presupunere a modelului. Timpul de acces la memorie diferă, de obicei, în funcție de locul de stocare a datelor, în *cache* (memoria volatilă) sau pe disc, ceea ce contrazice cea de-a treia presupunere. Dar chiar şi cu aceste neajunsuri, modelul *RAM* este un instrument excelent pentru a înțelege cum se comportă un algoritm în realitate, el reflectând comportarea computerului, dar rămânând destul de simplu pentru a putea fi manipulat şi înțeles.

Fiecare model are un interval-limită în care este util. Să considerăm ca exemplu *modelul Pământului plat*. Se poate afirma că acesta este un model incorect, pentru că este dovedit faptul că Pământul este rotund. Dar, pentru a construi fundația unei case, modelul Pământului plat este util şi poate fi utilizat ca atare. Este mult mai uşor de lucrat practic cu un astfel de model decât cu un model sferic. Tot astfel se întâmplă şi cu modelul de calcul *RAM*. El este,

de fapt, o abstractizare care, în general, devine foarte utilă pentru a explica performanța unui algoritm, într-un mod independent de mașină.

Complexitatea algoritmilor

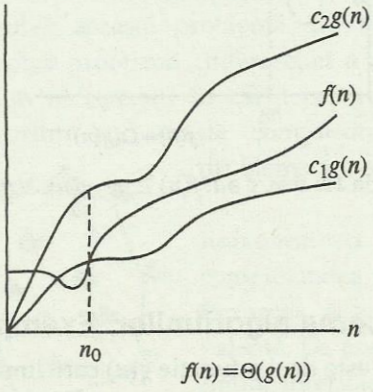


Prin *complexitatea* unui algoritm înțelegem de fapt *costul*, măsurat cu ajutorul unor anumiți parametri (timpul de execuție, memoria necesară, numărul anumitor operații etc.). Pentru a calcula complexitatea unui algoritm, avem nevoie să decidem care sunt acești parametri și să găsim o funcție cost corespunzătoare.

Dat fiind că resursele de memorie sunt, în practică, foarte mari, deducem că timpul de execuție al unui algoritm este parametrul de bază. Complexitatea timp poate fi calculată în funcție de numărul de operații elementare (unități de timp) necesare atunci când datele de intrare au o anumită dimensiune n. Așa cum am văzut mai sus, la descrierea modelului *RAM*, aceste operații de bază pot fi comparații (în număr foarte mare, de exemplu, în cazul unui algoritm de căutare), asignări (de exemplu, în cazul unui algoritm de sortare), adunări, înmulțiri, împărțiri, modulo, apeluri de metodă etc.

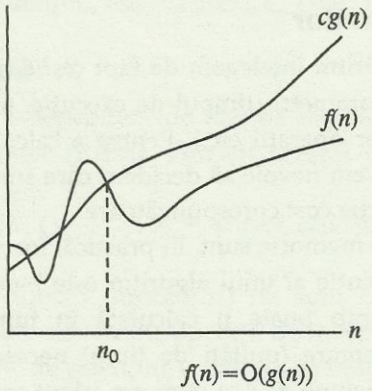
Notațiile Θ, O și Ω. În practică, sunt folosite o serie de notații care sunt utile pentru analiza performanței și a complexității unui algoritm. Aceste notații mărginesc valorile unei funcții f date cu ajutorul unor constante și al altei funcții. Prezentăm în continuare cele trei notații cunoscute în acest sens.

• **Notația Θ (categorie constantă – *same order*)**. Spunem că $f(n) = \Theta(g(n))$, dacă există constantele pozitive n_0, c_1 și c_2, astfel încât pentru toate numerele naturale n de la dreapta lui n_0 valoarea lui $f(n)$ se află întotdeauna între $c_1 \times g(n)$ și $c_2 \times g(n)$ inclusiv.



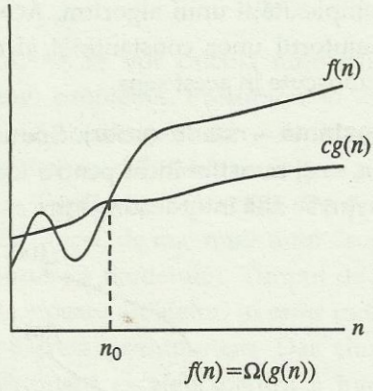
Notația Θ : $\exists n_0, c_1, c_2$, a.î. $c_1 \times g(n) \leq f(n) \leq c_2 \times g(n)$, $\forall n \geq n_0$.

• **Notația O (mărginire superioară – *upper bound*)**. Spunem că $f(n) = O(g(n))$, dacă există o constantă pozitivă n_0, astfel încât, pentru toate numerele n de la dreapta lui n_0, valoarea lui $f(n)$ se află întotdeauna sub $c \times g(n)$.



Notația O: $\exists n_0, c$, a.î. $f(n) \leq c \times g(n)$, $\forall n \geq n_0$.

• **Notația Ω (mărginire inferioară – *lower bound*)**. Spunem că $f(n) = \Omega(g(n))$, dacă există constantele pozitive n_0 și c, astfel încât, pentru toate numerele n mai mari decât n_0, valoarea lui $f(n)$ este mai mare decât $c \times g(n)$.



Notația Ω: $\exists n_0, c$ a.î. $f(n) \geq c \times g(n)$, $\forall n \geq n_0$.

Optimalitatea și reducerea algoritmilor. Exemple

Complexitatea unui algoritm este deci o funcție $g(n)$ care limitează superior numărul de operații (la execuție) necesare pentru dimensiunea n a problemei. Există două interpretări ale limitei superioare:

a) complexitatea în cazul cel mai defavorabil (*worst-case complexity*): timpul de execuție pentru orice dimensiune dată va fi mai mic decât limita superioară, exceptând câteva dimensiuni ale problemei, unde este atins un maxim;
b) timpul de execuție pentru orice dimensiune dată va fi media numărului de operații pentru toate instanțele posibile ale problemei.

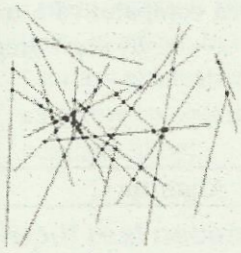
Deoarece este dificil să se estimeze o comportare statistică ce depinde de dimensiunea intrării, de cele mai multe ori este folosită prima interpretare, cea a cazului celui mai defavorabil. În majoritatea cazurilor, complexitatea lui $f(n)$ este aproximată de familia sa $O(g(n))$, unde $g(n)$ este una dintre funcțiile: n (complexitate liniară), $log(n)$ (complexitate logaritmică), n^a, $a \geq 2$ (complexitate polinomială), a^n (complexitate exponențială), $n!$ (complexitate factorială). O altă definiție sugestivă echivalentă este cea folosind limitele din analiza matematică: dacă limita funcției $f(n)/g(n)$ există și este finită, atunci $f(n)$ are ordinul $O(g(n))$.

Exemple:

1. $lim\ (sqrt(n)/n) = 0 \rightarrow$ sqrt$(n) = O(n)$
2. $lim\ (n/sqrt(n))$ = infinit $\rightarrow n$ nu este $O(sqrt(n))$
3. $lim\ (n/2n) = ½ \rightarrow n = O(2n)$
4. $lim\ (2n/n) = 2 \rightarrow 2n = O(n)$

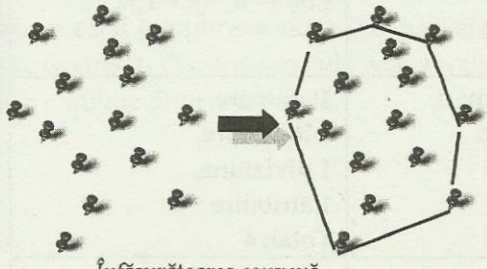
Pentru n suficient de mare au loc inegalitățile: $log(n) < n < n \times log(n) < n^2 < n^3 < 2^n$, ceea ce implică $O(log(n)) < O(n) < O(n \times log(n)) < O(n^2) < O(n^3) < O(2^n)$.

Exemple: $10n + 5\ log(n) + 8 \in O(n)$, $8n \in O(n^2)$, $65 \in O(1)$, $n^{1000} \in O(2^n)$.



Intersecția a n *segmente*

Odată determinată complexitatea unui algoritm, apare problema stabilirii dacă acest algoritm este *optimal*. Un algoritm pentru o problemă dată este optimal atunci când complexitatea sa atinge o limită inferioară a tuturor algoritmilor ce rezolvă această problemă. De exemplu, orice algoritm ce rezolvă problema „intersecției a n segmente" va executa cel puțin n^2 operații în cazul cel mai nefavorabil și există un algoritm cu această complexitate, deci vom afirma că problema are o complexitate $\Omega(n^2)$.



Înfășurătoarea convexă

Reducerea este o altă tehnică de a estima complexitatea unei probleme prin transformarea acesteia într-o problemă echivalentă. De exemplu, presupunând că știm limita inferioară a unei probleme A și că ne dorim estimarea limitei inferioare a

unei probleme B. Dacă putem transforma A în B prin intermediul unui pas de transformare al cărui cost este mai mic decât rezolvarea lui A, atunci B are aceeaşi limită ca şi A. Ca un exemplu, putem considera problema determinării înfăşurătorii convexe, care se reduce la sortarea punctelor date (complexitate: $\Theta(n \times log(n))$).

Exemple: $O(n) + O(log(n)) = O(n)$ $(log(n) < n)$, $O(n!) + O(n^3) = O(n!)$ $(n^3 < n!)$.

Exemplul 1. Analiza complexității. Considerăm următorul algoritm:

```
1. Citeşte n
2. dif ← 0, sum ← 0, k ← 0
3. While (k < n) Do
                sum ← sum + 1
                dif ← dif – 1
     k ← k + 1
   End_While
4. k ← 0
5. While (k < 3n) Do
                sum ← sum – 1
                k ← k + 1
   End_While
6. Scrie sum, dif
```

Analiza complexității:
Pasul 2 are nevoie de 3 unități de bază.
Ciclul de la pasul 3 va executa $2n$ adunări, n scăderi şi $3n$ atribuiri, deci în total $6n$ unități de timp.
Pasul 4 are nevoie de o unitate.
Pasul 5 face $2 \times 3n$ atribuiri, $3n$ scăderi şi $3n$ adunări, deci în total $12n$ unități de bază.

Total: $3 + 6n + 1 + 12n = 18n + 4$

Aceasta înseamnă $O(n)$.

Exemplul 2. Analiza complexității şi a eficienței. Vom face o analiză comparativă a trei algoritmi ce rezolvă aceeaşi problemă: calculul sumei $1 + 2 + ... + n$. În acest scop, putem scrie următorii algoritmi:

Analiza eficienței algoritmilor

Algoritm A	Algoritm B	Algoritm C

sum ← 0
pentru (*i* ← 1, *n*)
  sum ← *sum* + *i*
	sum ← 0
	pentru (*i* ← 1, *n*){
	  pentru (*j* ← 1, *i*)
	  sum ← *sum* + 1
	}
		sum ← *n* * (*n* + 1)/2

$n + 1$ asignări, n atribuiri **Total: $2n + 1$**	$1 + n(n + 1)/2$ asignări, $n(n + 1)/2$ atribuiri **Total: $n^2 + n + 1$**	1 adunare, 1 înmulțire, 1 diviziune, 1 atribuire **Total: 4**
$O(n)$	$O(n^2)$	$O(1)$

Deducem că algoritmul cel mai performant este *C* (cunoştinţele de matematică sunt importante în conceperea algoritmilor!), fiind şi algoritmul optimal pentru aflarea acestei sume, iar cel mai ineficient este *B*.

Exemplu comparativ de creştere a lui *O(g(n))*

Rata comparativă de creştere a funcţiei O(g(n))

n	$\log(\log n)$	$\log n$	$(\log n)^2$	n	$n \log n$	n^2	2^n	$n!$
10	2	3	11	10	33	10^2	10^3	10^5
10^2	3	7	44	100	664	10^4	10^{30}	10^{94}
10^3	3	10	99	1000	9966	10^6	10^{301}	10^{1435}
10^4	4	13	177	10.000	132.877	10^8	$10^{3.010}$	10^{19335}
10^5	4	17	276	100.000	1.660.964	10^{10}	$10^{30.103}$	$10^{243.338}$
10^6	4	20	397	1.000.000	19.931.569	10^{12}	$10^{301.030}$	$10^{2.933.369}$

Alte complexităţi:

1. Se poate demonstra că *algoritmul lui Euclid* are complexitatea $O((\log a)(\log b))$.
2. Complexitatea pentru *căutarea liniară* este $O(n)$, deoarece, în cel mai defavorabil caz, trebuie să parcurgem toată secvenţa de numere, deci complexitatea este liniară.
3. Pentru *căutarea binară*, complexitatea este $O(\log n)$; în cazul cel mai nefavorabil avem nevoie de $(1 + \log n)$ iteraţii pentru a găsi elementul x.
4. Pentru *problema colorării hărţilor*, unde avem un graf cu n vârfuri şi trebuie atribuite m culori, în cazul cel mai defavorabil, vor trebui generate toate combinaţiile (căutare exhaustivă), care sunt m^n (numărul funcţiilor definite pe o mulţime cu n elemente cu valori într-o mulţime cu m elemente), deci complexitatea este exponenţială: $O(m^n)$.

Timpul real necesar unui algoritm (polinomial *vs* exponenţial)

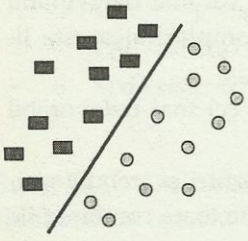
Următoarea tabelă arată cum se modifică timpul real necesar sistemului de calcul atunci când trebuie executaţi $T(n)$ paşi simpli pe un computer obişnuit, cu 1 bilion de paşi/secundă. O microsecundă este a milioana parte dintr-o secundă şi o milisecundă este o miime dintr-o secundă.

Analiza comparativă a timpului real necesar
(algoritmi polinomiali vs algoritm exponențial)

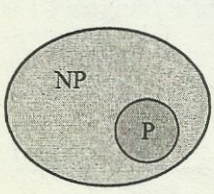
n	$T(n) = n$	$T(n) = n \lg(n)$	$T(n) = n^2$	$T(n) = n^3$	$T(n) = 2^n$
5	0.005 microsec.	0.01 microsec.	0.03 microsec.	0.13 microsec.	0.03 microsec.
10	0.1 microsec.	0.03 microsec.	0.1 microsec.	1 microsec.	1 microsec.
20	0.02 microsec.	0.09 microsec.	0.4 microsec.	8 microsec.	1 milisec.
50	0.05 microsec.	0.28 microsec.	2.5 microsec.	125 microsec.	13 zile
100	0.1 microsec.	0.66 microsec.	10 microsec.	1 milisec.	4×10^{13} ani

Remarcăm că, pentru $n \geq 50$, timpul de calcul pentru $T(n) = 2^n$ devine prea mare pentru a fi şi practic. Chiar dacă se măreşte de un milion de ori viteza sistemului, pentru $n = 100$ tot va fi nevoie de 40.000.000 de ani pentru a afla răspunsul.

Clasificarea problemelor (*P*, *NP*, *NP*-complete, *NP-hard*)

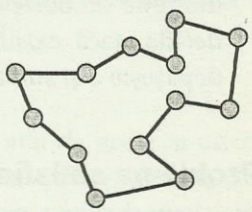


Pe lângă determinarea complexității timp, ne putem referi şi la clasificarea problemelor după criteriul dat de faptul că ele sunt „greu" sau „uşor" de rezolvat (accesibile sau inaccesibile). Această schemă de clasificare include clasele *P* şi *NP*, iar „*NP*-complete" şi „*NP-hard*" sunt subclase ale clasei *NP*. În vederea stabilirii unor caracteristici formale relativ la mulțimea problemelor posibile, a fost mai întâi definită clasa *problemelor de decizie*. Această clasă conține toate acele probleme a căror soluție poate fi „Da" sau „Nu". De exemplu, problema deciziei conexității unui graf unidirecțional (adică dacă între oricare două vârfuri ale grafului există un drum). Datele de intrare sunt vârfurile şi muchiile grafului *G* (elemente care determină graful), iar întrebarea este următoarea: „Este graful *G* conex?". De obicei, cele mai multe probleme de optimizare nu sunt probleme de decizie, dar se pot transforma uşor în astfel de probleme. De exemplu, una dintre versiunile problemei comisului voiajor (*TSP – Travelling Salesman Problem*) are ca date de intrare graful *G* cu costurile muchiilor şi un număr *K*. Întrebarea asociată este: „*G* conține un tur de lungime mai mică sau egală cu *K*?". De obicei, o problemă de optimizare nu este mult mai greu de rezolvat decât problema de decizie corespunzătoare.

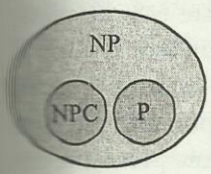


Clasa *P* se defineşte ca fiind mulțimea problemelor de decizie pentru care există un algoritm în timp polinomial de rezolvare, adică acele probleme care sunt considerate formal „uşoare" (accesibile).

Mai larga clasă de probleme *NP* conține clasa *P* şi se referă la probleme cu algoritmi de rezolvare „nedeterministic-polinomiali". Clasa *NP* conține acele probleme de decizie care au următoarea proprietate: pentru orice instanță „da" a problemei, rezultatul se poate verifica folosind un algoritm de tip polinomial. De exemplu, dacă considerăm problema: „Sunt grafurile *G* şi *G'* izomorfe?", „instanța da" a acestei probleme este o instanță în care grafurile *G* şi *G'* sunt izomorfe. O demonstrație a acestei instanțe va fi un izomorfism (o mapare 1-1) *f* de la *G* la *G'*. Demonstrarea răspunsului presupune, de fapt, verificarea faptului că *f* este într-adevăr un izomorfism de la *G* la *G'*, adică, pentru orice pereche de muchii $\{u, v\}$ din *G*, $\{f(u), f(v)\}$ este muchie în *G'* şi *G'* nu are alte muchii în afară de cele verificate. Această demonstrație se poate realiza în timp polinomial relativ la dimensiunea grafurilor *G* şi *G'*, deci această problemă aparține clasei *NP*. Un alt exemplu este şi problema comisului voiajor, prezentată anterior în varianta de decizie, unde instanța este un graf *G* şi un număr $K = 100$. Dacă răspunsul este „da", atunci există o listă de muchii de lungime cel mult 100 care este un circuit; această listă acționează ca un „certificat" şi se poate demonstra în timp polinomial că este o soluție validă, fapt suficient pentru a decide că răspunsul este „da". Nu există simetrii între cele două variante „da" şi „nu". Dacă, pentru a ne convinge că răspunsul este „da", este suficient să verificăm în timp polinomial o variantă de circuit, în cazul răspunsului „nu", totul devine mult mai complicat (ar trebui să verificăm toate variantele posibile, ceea ce este extrem de costisitor, practic, imposibil). De fapt, prin schimbarea rolurilor jucate de „da" şi „nu", se obține o clasă de probleme numită *Co-NP* (de exemplu, pentru problema comisului voiajor: „toate circuitele posibile ale comisului voiajor în *G* au lungimea mai mare decât *K*?"). Există şi multe probleme ce par să se situeze în afara claselor *NP* şi *Co-NP*, pentru că ele nu posedă o demonstrație în timp polinomial („certificat"). Un exemplu poate fi problema cu datele de intrare graful *G* şi numerele *K* şi *L*, cu întrebarea: „Este numărul circuitelor distincte ale comisului voiajor, de lungime cel mult *K*, egal cu *L*?".



Probleme *NP*-complete



Până acum, nu s-a găsit un algoritm polinomial pentru rezolvarea problemei comisului voiajor. Pe de altă parte, nimeni nu a fost capabil să demonstreze că un astfel de algoritm nu există. Deci se poate afirma că problema comisului voiajor, ca şi multe alte probleme, este „intratabilă" (inaccesibilă). Se poate demonstra că varianta de decizie a problemei comisului voiajor, ca şi multe alte probleme din clasa *NP*, sunt cele mai *grele* probleme *NP*, în sensul că, dacă există un algoritm în timp polinomial pentru *una* dintre aceste probleme, atunci există un algoritm polinomial

pentru *toate* problemele din *NP*. Observăm că aceasta este o afirmație foarte puternică, de vreme ce *NP* conține un număr imens de probleme care par să fie extrem de dificil de rezolvat atât teoretic, cât și practic! Problemele din *NP* cu această proprietate se numesc *NP*-complete.

Alte exemple de probleme *NP*-complete :

- problema determinării dacă două grafuri date sunt izomorfe;
- problema colorării grafurilor (*Graph Colouring Problem*): date fiind un graf G și un număr natural $k \geq 1$, există o k-colorare a lui G ? (k-colorare înseamnă posibilitatea de a se eticheta vârfurile grafului cu culorile 1, ..., k, astfel încât oricare două vârfuri vecine să fie colorate diferit);
- problema orarului (*Schedule Problem*): decizia existenței unui orar pentru un set de obiective, resurse și caracteristici ale unor activități;
- problema împachetării (*Bin Packing Problem*): date fiind o mulțime de obiecte cu dimensiunile lor și o mulțime de lăzi, decizia dacă obiectele pot fi împachetate în lăzi;
- problema partiționării (*Partition Problem*) unei mulțimi de numere întregi în două submulțimi, astfel încât suma numerelor din primul grup este egală cu suma numerelor din al doilea grup;
- problema acoperirii mulțimilor (*Set Cover Problem*): dată fiind o mulțime S, o colecție de submulțimi ale lui S și un număr natural k, să se decidă dacă există posibilitatea de a scrie pe S ca reuniune de k sau mai puține submulțimi din colecție;
- problema rucsacului (*Knapsack Problem*): date fiind un rucsac de dimensiune S, o mulțime de obiecte cu dimensiunile și greutățile lor și un număr întreg V, să se decidă dacă există o mulțime de obiecte pentru care suma dimensiunilor nu depășește S și suma greutăților este V sau mai mare.

Problema satisfiabilității (*SAT*)

Clasa *NP* și noțiunea de *probleme „complete"* din *NP* au fost prima dată introduse de Cook într-un articol din 1971. În articolul respectiv, el a demonstrat că o problemă particulară de decizie din logică, problema satisfiabilității (*SAT*), este *NP*-completă, dar și că orice altă problemă din *NP* poate fi codificată ca fiind un caz special al problemei *SAT*. Odată rezolvată o problemă *NP*-completă, orice problemă *NP*-completă devine rezolvabilă, prin furnizarea unei transformări polinomiale între cele două. Putem afirma deci că *SAT* este una dintre cele mai importante probleme din teoria complexității.

Considerându-se o formulă logică în forma normală conjunctivă (*CNF*), să se decidă dacă există o asignare a variabilelor, astfel încât evaluarea formulei să aibă valoarea adevărat. O formulă logică este în formă normală conjunctivă (*FNC*) dacă are forma: $(y_{11} \vee y_{12} \vee ... \vee y_{1n_1}) \wedge (y_{21} \vee y_{22} \vee ... \vee y_{2n_2}) \wedge ... \wedge (y_{m1} \vee y_{m2} \vee ... \vee y_{mn_m})$, unde y_{ij} sunt valori boolene din mulțimea de variabile $\{x_1, x_2, ..., x_n, \neg x_1, \neg x_2, ..., \neg x_n\}$.

Exemplu

Presupunem următoarea formulă în *FNC*:

$$F = (x_1 \vee \neg x_2) \wedge (x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_3)$$

Problema este de a decide dacă există o asignare a variabilelor x_1, x_2, x_3, astfel încât valoarea de adevăr a lui F corespunzătoare ei să fie 1. Observăm că în acest caz răspunsul este da, existând două astfel de asignări.

x_1	x_2	x_3	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Clasa problemelor *NP-hard*

Clasa problemelor *NP-hard* este clasa problemelor cel puțin tot atât de grele ca orice problemă din *NP*. În particular, problemele de optimizare pentru care problema de decizie corespunzătoare este *NP*-completă (de exemplu, problema comisului voiajor) sunt *NP-hard*, deoarece versiunea de optimizare este cel puțin tot atât de dificilă ca și versiunea de decizie. Am văzut și mai sus că timpul real necesar găsirii unei soluții prin căutare exhaustivă, de exemplu, este inacceptabil; de aceea, găsirea unei soluții optime este imposibilă. În ultimii ani, au apărut domenii ale informaticii care încearcă găsirea unor soluții „cât mai optime": algoritmii evolutivi, inteligența artificială, sistemele *fuzzy*, toate fiind inspirate de natură.

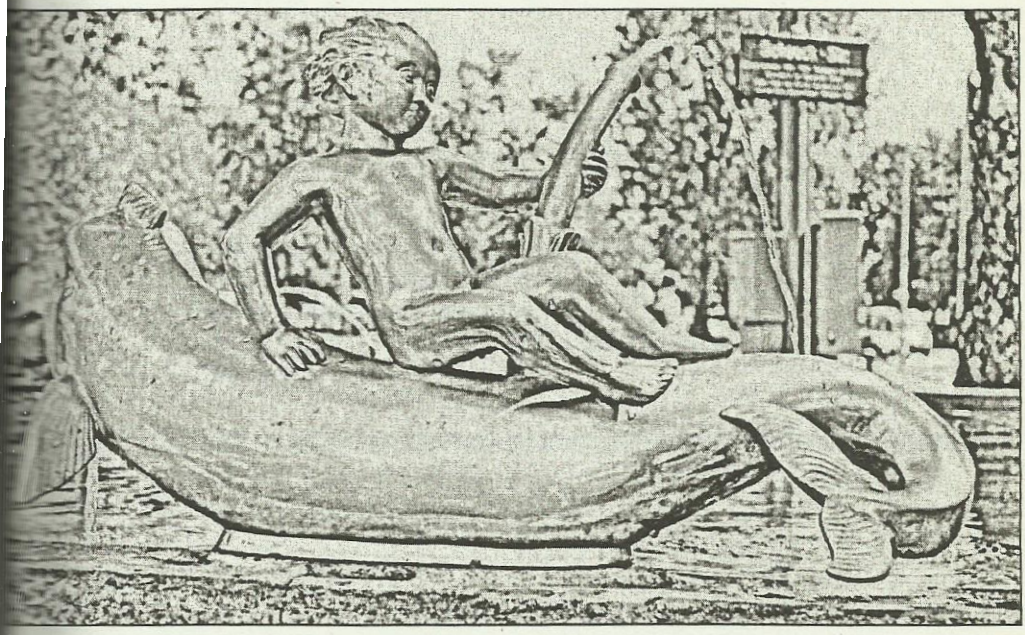
Exerciții și probleme propuse

1. Scrieți un referat despre viața și contribuțiile lui Eratostene (Erathostenes) sau Euclid, folosind resurse de pe internet.

2. De unde provine cuvântul *algoritm*? Dați câteva definiții alternative.
3. Scrieți formal „algoritmul lui Euclid" și transformarea variabilelor pentru a afla cel mai mare divizor comun al numerelor 2322 și 654.
4. Scrieți formal algoritmul „sita lui Eratostene" și aplicați-l practic pentru a afla numerele prime mai mici decât 30.
5. Scrieți formal algoritmul de înmulțire a numerelor întregi.
6. Construiți formal algoritmul de determinare simultană a minimului și maximului dintr-o secvență de elemente. Care este complexitatea sa?
7. Scrieți algoritmii de căutare liniară, respectiv binară. Ce complexități au? Care este mai performant? Cum se explică acest lucru?
8. De ce putem afirma că o rețetă culinară este tot un algoritm?
9. Ce sunt *reprezentarea* și *transformarea* în cadrul ciclului de viață al unui program? Cum se codifică *problema colorării hărților*?
10. Care sunt proprietățile unui algoritm?
11. Ce înțelegem prin *proiectarea*, respectiv *analiza* algoritmului?
12. Ce se înțelege prin *model de calcul*? Dați exemple de modele de calcul. Descrieți proprietățile și avantajele modelului *RAM*.
13. Ce este *complexitatea* unui algoritm? Descrieți formal și grafic notațiile Θ, O și Ω.
14. Ce înțelegem prin *reducerea*, respectiv *optimalitatea* unui algoritm ?
15. Cum se clasifică problemele relativ la complexitate? Dați exemple de probleme *NP*-complete.
16. Descrieți problema *SAT*. Dați și un exemplu sugestiv. De ce este ea considerată cea mai importantă problemă *NP*-completă?
17. Care este clasa problemelor *NP-hard*?
18. Care dintre următoarele afirmații sunt adevărate?

    A) $n^2 \in O(n^3)$ B) $n^3 \in O(n^2)$ C) $2^{n+1} \in O(2^n)$ D) $(n+1)! \in O(n!)$
19. Determinați ordinul următorilor algoritmi:

pentru $i \leftarrow 1, n$ **execută** **pentru** $j \leftarrow 1, 5$ **execută** {operație elementară}	**pentru** $i \leftarrow 1, n$ **execută** **pentru** $j \leftarrow 1, i+1$ **execută** {operație elementară}
pentru $i \leftarrow 1, n$ **execută** **pentru** $j \leftarrow 1, 6$ **execută** **pentru** $k \leftarrow 1, n$ **execută** {operație elementară}	**pentru** $i \leftarrow 1, n$ **execută** **pentru** $j \leftarrow 1, i$ **execută** **pentru** $k \leftarrow 1, n$ **execută** {operație elementară}



Detaliu al unei fântâni publice din Zürich

Capitolul 2
Introducere în POO

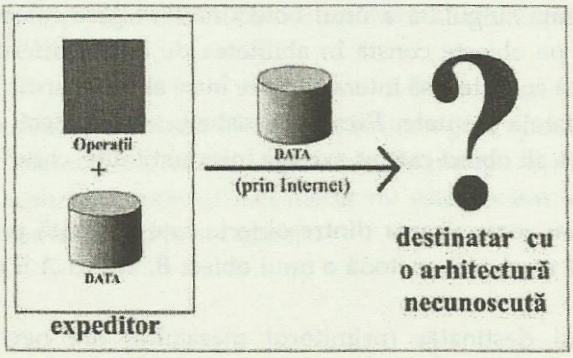
- Concepte de bază
- Proprietăţi ale programării orientate obiect
- Crearea şi distrugerea obiectelor
- Elemente statice ale obiectelor
- Avantajele utilizării programării orientate obiect
- Probleme propuse

When you discover a beautiful place like Positano
the first impulse you have is not to tell to anybody else your discovery.
You think: if I tell about this place to other people it will fill with tourists that will spoil it.
John Steinbeck

Concepte de bază

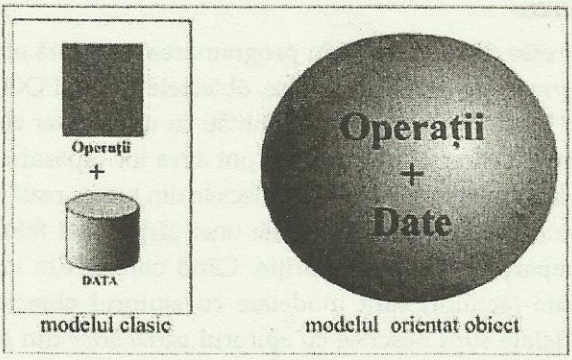
Conceptul de *obiect* este cheia de boltă în programarea orientată obiect (POO; în engl., *Object Oriented Programming – OOP*). În mare, obiectele soft în POO descriu obiecte din lumea reală, cum ar fi case, animale, clădiri; entități cu un caracter abstract, de genul formulelor matematice sau chiar evenimente ce pot avea loc (apăsarea unei taste, producerea unei explozii sau soldarea unui cont). Obiectele din lumea reală sunt caracterizate de o mulțime de atribute. De exemplu, atributele unei cărți pot fi titlul, autorul, formatul, editura, culoarea coperții și anul de apariție. Când obiecte din lumea reală, abstracțiuni sau evenimente (acțiuni) sunt modelate cu ajutorul obiectelor soft, atributele acestor entități modelate sunt descrise cu ajutorul *variabilelor* din obiectul soft respectiv. Programarea structurată clasică (neorientată obiect sau procedurală) se bazează pe un model de programare ce separă operațiile (funcții, proceduri etc.) de datele asupra cărora operațiile acționează. Rigiditatea acestui tip de programare reiese din controlul dificil al schimbărilor ce pot avea loc atât în structura datelor, cât și în cea a operațiilor.

Problema se complică și mai mult dacă se ia în calcul funcționarea programelor într-un mediu distribuit, în care arhitecturile hard și soft sunt foarte diverse. De cele mai multe ori, expeditorul unei informații (într-un mediu distribuit, cum ar fi internetul) nu știe aproape nimic despre arhitectura mașinii destinatar, excluzându-se astfel posibilitatea existenței unui „translator" care să traducă datele destinatarului, astfel încât acesta să poată să le „înțeleagă" (vezi figura de mai jos). Soluția la această problemă o oferă modelul de programare orientat obiect, în care datele și operațiile care le prelucrează sunt grupate în *obiecte*.



Transferul datelor prin internet

Deoarece datele şi operațiile sunt plasate într-o singură entitate (numită obiect), comunicarea într-un mediu distribuit are loc mult mai uşor, practic, ea rezumându-se la transportul dintr-un loc în altul al obiectelor. Accesul destinatarului la date se va face prin operațiile furnizate de obiect. Uşurința acestui tip de comunicare la distanță este datorată existenței multor tehnologii sau standarde industriale ce permit comunicarea între obiecte aflate pe platforme diferite (exemplu: *IIOP* definit de *Object Management Group*).



După cum am precizat şi mai înainte, datele sunt stocate în obiecte cu ajutorul variabilelor. Operațiile ce lucrează cu aceste date (funcții, proceduri etc.) poartă numele de *metode*. Un obiect poate fi văzut ca o mulțime de variabile şi metode. În general, este recomandat ca accesul la datele (variabilele) unui obiect să nu fie permis decât metodelor obiectului respectiv, realizându-se astfel o protecție sporită a datelor interne. Această proprietate a programării orientate obiect se numeşte *încapsulare*. *Exemplu*: La datele obiectului „carte" nu se poate ajunge decât prin intermediul metodelor dedicate (conținutul variabilei „pagini" nu se poate determina decât cu ajutorul meodei „nr_pagini()"). În C++ şi Java, ascunderea datelor se face cu modificatorii de acces „private" sau „protected".

Un obiect este, de obicei, o componentă dintr-un program sau o aplicație, alături de alte obiecte, existența singulară a unui obiect nefiind prea folositoare. Tot secretul metodei orientate pe obiecte constă în abilitatea de a face diferite obiecte – fiecare descriind o anumită entitate – să interacționeze între ele şi în urma producerii acestora să se obțină rezultatele scontate. *Exemplu*: existența unui obiect „carte" nu are sens decât dacă există un alt obiect care să extragă informații din „carte", apelând o metodă de tipul „citeşte()".

Interacțiunea sau comunicarea dintre obiecte este realizată prin *mesaje*. Dacă un obiect *A* doreşte să execute o metodă a unui obiect *B*, atunci *A* îi trimite obiectului *B* un mesaj.

Uneori, obiectul destinatar (primitorul mesajului) are nevoie de mai multe informații pentru a executa metoda cerută de expeditorul mesajului (solicitantul).

Exemplu: obiectul „student" trimite un mesaj obiectului „carte" prin care îi cere acestuia să apeleze metoda „deschide_carte()". Această metodă a obiectului „carte" nu se poate executa, deoarece „student"-ul nu a precizat „cărții" pagina la care trebuie să se deschidă.

Aceste informații suplimentare, de care este nevoie pentru execuția unor metode, se numesc *parametri de intrare*. Pentru ca metoda „deschide_carte()" din obiectul „carte" să fie executată, trebuie ca expeditorul (în cazul nostru „student"-ul) să trimită în mesaj şi valori pentru toți parametrii de intrare ai metodei ce se doreşte să fie apelată ("student"-ul trebuie să trimită în mesaj şi valorile parametrului de intrare „pagina", pentru a nu exista ambiguități vizavi de pagina la care va fi deschisă cartea).

O metodă poate avea zero, unul sau mai mulți parametri de intrare.

Unele metode, în urma execuției lor, pot furniza date numite *parametri de ieşire*. O metodă poate avea sau nu parametri de ieşire.

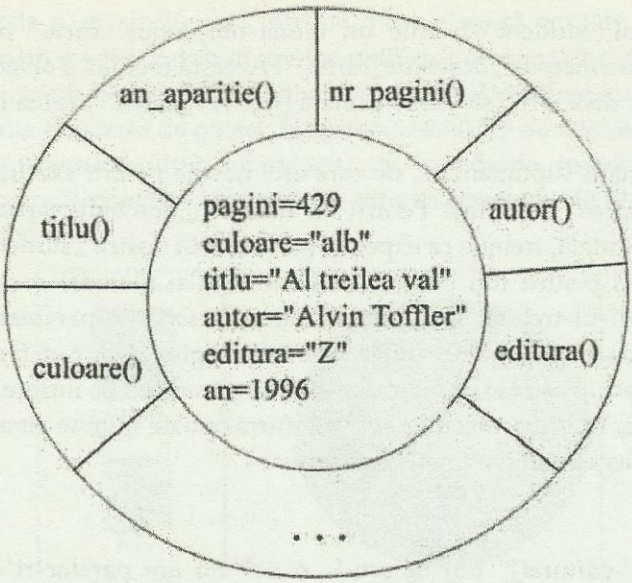
Exemple:

- Metoda „editura()" din obiectul „carte" nu are parametri de intrare şi ca parametru de ieşire întoarce valoarea variabilei „editura".
- Metoda „citeşte_pagina()" primeşte ca parametru de intrare numărul paginii şi ca parametru de ieşire întoarce conținutul paginii al cărui număr a fost primit ca parametru de intrare.
- Metoda „deschide_carte" primeşte ca parametru de intrare numărul paginii şi nu are parametri de ieşire, deoarece, în urma execuției metodei, nu este întoarsă nici o valoare (are loc doar operațiunea de deschidere a cărții) .
- Metoda „închide_carte" nu are nevoie nici de parametrii de intrare, nici de cei de ieşire.

Un *mesaj* trimis unui obiect este compus din trei elemente:

1. numele obiectului care conține metoda;
2. numele metodei;
3. valori pentru toți parametrii de intrare ai metodei (dacă metoda are parametri de intrare).

Deoarece tot ce poate să facă un obiect este realizat prin apelarea metodelor sale, interacțiunea dintre obiecte are loc prin schimburi de mesaje. Obiectele nu trebuie să aparțină neapărat aceluiaşi proces şi nici măcar nu este necesar să existe pe aceeaşi maşină pentru a putea trimite şi primi mesaje.



O *clasă* este schema (planul) după care este creat un obiect. În interiorul unei clase, sunt precizate denumirile tuturor variabilelor şi tipurile de valori pe care acestea le vor stoca (numeric, şir de caractere etc.), alături de toate operaţiile (metodele) pe care un obiect, construit din respectiva clasă, le va conţine.

Operaţia prin care este creat un obiect, după schema precizată de o clasă se numeşte *instanţiere*. Obiectul creat după planul oferit de o clasă este o *instanţă* a acelei clase. Nu se poate crea un obiect fără a se declara clasa (schema) după care acesta a fost instanţiat.

În lumea reală, există foarte multe obiecte asemănătoare, descrise de aceeaşi mulţime de caracteristici. Cărţile, de exemplu, au toate titlu, autor, editură, an de apariţie, format şi pot fi deschise, închise, citite etc. Dacă se doreşte crearea unei aplicaţii care să ia în considerare o mulţime de cărţi, primul pas ar fi crearea unei clase „carte" în care sunt date denumirile variabilelor, tipul valorilor suportate de ele şi metodele corespunzătoare. Pasul următor ar fi instanţierea obiectelor de tip „carte", fiecare în parte având valori specifice pentru variabilele sale (pentru autor, titlu, editură etc.).

Cu alte cuvinte, în programarea orientată obiect trebuie respectată următoarea ordine:

1. crearea claselor:
   - precizarea denumirii variabilelor, a tipului de date suportat de fiecare variabilă în parte (tip standard: întreg, real, şir de caractere, etc. sau derivat: o matrice, o altă clasă etc.), a domeniului de vizibilitate al variabilei (în general, este recomandată respectarea proprietăţii de încapsulare, prin

folosirea de variabile având modificatorul de acces setat pe *private* sau *protected*, astfel, accesul la variabile nefiind permis decât metodelor ce fac parte din aceeaşi clasă cu variabila) etc.;

- precizarea signaturii metodelor componente (a denumirii şi a parametrilor de intrare, respectiv de ieşire), a corpurilor lor (codul care descrie comportamentul metodei), a domeniului de vizibilitate şi a altor proprietăţi mai mult sau mai puţin specifice unei anumite implementări a metodei orientate obiect.

2. crearea obiectelor, prin specificarea clasei după a cărui model a fost creat obiectul respectiv şi prin asignarea de valori variabilelor precizate în definiţia clasei.

Doar în momentul instanţierii unei clase (a naşterii unui obiect) variabilele definite de clasă sunt alocate în memorie şi pot fi prelucrate de către metode.

Fiecare instanţă (obiect) a unei clase are propria mulţime de valori, corespunzătoare variabilelor sale. De exemplu, fiecare carte are autorul şi titlul proprii.

Toate instanţele unei clase suportă doar metodele precizate în definiţia clasei, singura diferenţă între obiectele provenite din aceeaşi clasă constând în valorile diferite (dar de acelaşi tip) ce sunt stocate în variabile.

Proprietăţi ale programării orientate obiect

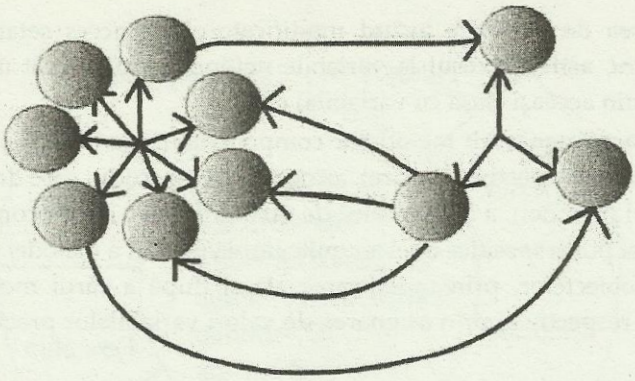
1. *Încapsularea*

Accesul la variabilele unui obiect se face doar prin apelarea metodelor sale, orice alt obiect neştiind nici măcar de existenţa variabilelor obiectului în cauză.

2. *Moştenirea*

Unul dintre marile avantaje ale programării orientate obiect este posibilitatea de a reutiliza codul existent. Posibilitatea proiectării de noi clase folosind clase deja construite se numeşte *moştenire*. Dacă o clasă *A* moşteneşte o clasă *B*, atunci variabilele şi metodele clasei *B* vor fi considerate ca aparţinând şi clasei *A*. Moştenirea permite crearea unor clase „de bază", cu rolul de a stoca trăsături comune unor clase diferite; astfel, aceste proprietăţi nu vor trebui precizate în fiecare clasă în parte.

Exemplu. Clasele „triunghi", „patrulater" şi „cerc" au structuri diferite, dar toate au o arie, un număr de laturi, perimetru etc. Pentru ca în cele trei clase să nu se repete aceleaşi elemente, variabile şi metode, soluţia optimă este proiectarea unei clase „figură geometrică", în care să se regăsească trăsăturile comune şi aceasta să fie moştenită de toate cele trei clase în cauză.

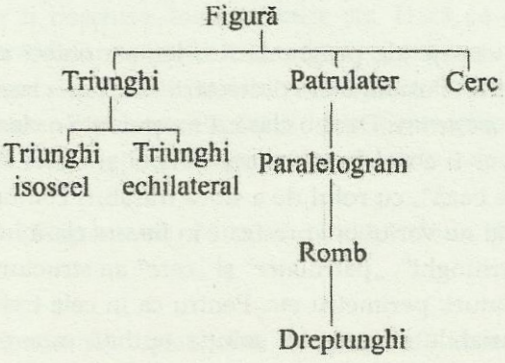


După cum se vede şi din imaginea de mai sus, moştenirea este posibilă şi pe mai multe niveluri. A nu se confunda moştenirea unei clase cu instanțierea ei. Numai o clasă poate moşteni proprietățile altei clase. În urma instanțierii unei clase, rezultă un obiect.

În urma moştenirii unei clase, clasa derivată (care moşteneşte proprietățile clasei de bază) nu copie proprietățile clasei de bază, ci acestea îi sunt direct accesibile.

Datorită proprietății de moştenire, o clasă derivată nu numai că are acces la variabilele şi metodele clasei moştenite, dar le şi poate redefini. Datorită acestui fapt, o practică obişnuită în proiectarea orientată pe obiecte este declararea de metode cu corpul vid în clasele de bază, urmând ca acestor metode să li se ofere un corp în clasele derivate.

Exemplu. Clasa „figura geometrică" poate conține o metodă „aria()" care să calculeze aria figurii, dar o implementare a acesteia nu poate fi dată decât în clasele derivate (formula de calcul al ariei este diferită, în funcție de tipul figurii).



Moştenirea uşurează munca proiectanţilor de sisteme informatice prin modularitatea pe care o promovează; astfel, situaţii complexe din lumea reală pot fi mai uşor modelate.

3. *Polimorfismul*

Abilitatea unei metode cu un anumit nume de a avea comportări diferite în funcţie de parametrii de intrare se numeşte *polimorfism*. Termenul *polymorph* provine din greceşte, limbă în care înseamnă „a avea mai multe forme". Nu trebuie să se confunde această proprietate cu cea de moştenire, cu ajutorul căreia se pot redefini metodele moştenite de la o altă clasă. Polimorfismul se referă la metode cu acelaşi nume şi cu parametri de intrare şi/sau ieşire diferiţi. În ceea ce priveşte moştenirea, este vorba despre metode redefinite, dar care îşi păstrează aceeaşi signatură (acelaşi nume şi parametri de intrare/ieşire).

Polimorfismul are un rol foarte important în reutilizarea codului. Să presupunem că avem o clasă „operaţii aritmetice" cu o metodă care primeşte ca parametri două numere oarecare (reale sau complexe). Dacă polimorfismul nu ar fi luat în calcul, atunci ar trebui implementate metode, fiecare cu o denumire diferită, pentru toate cele patru cazuri (număr real + număr real, număr real + număr complex, număr complex + număr real şi număr complex + număr complex) şi utilizatorul clasei ar trebui să ţină minte toate cele patru nume de metode, pentru a face o adunare. Cu ajutorul polimorfismului, cele patru metode distincte pot avea acelaşi nume (în C++, de exemplu, supraîncărcarea operatorului +), iar programatorul care utilizează clasa „operaţii aritmetice" nu trebuie să reţină decât o singură denumire, compilatorul alegând varianta corectă în funcţie de parametrii de intrare furnizaţi.

Pe scurt, prin polimorfism, o metodă poate executa acţiuni diferite, în funcţie de parametrii de intrare care îi sunt furnizaţi. Se realizează o interfaţă de comunicare a obiectului cu exteriorul mult mai unitară şi mai flexibilă (metoda „adunare" are un singur nume, şi nu patru), proiectanţii putând să se concentreze mai bine asupra modului cum să folosească obiectele, fiind scutiţi de detalii de implementare.

În unele limbaje de programare ce suportă metoda orientată pe obiecte, cum ar fi C++, este permisă supraîncărcarea operatorilor standard predefiniţi. De exemplu, operatorul „+", destinat efectuării operaţiei de adunare pentru numere reale, poate fi extins să execute şi alte tipuri de adunări.

Crearea şi distrugerea obiectelor

Obiectele soft, ca şi obiectele din lumea reală, au o viaţă limitată. Ele sunt create (le sunt alocate resurse), folosite şi distruse (resursele alocate sunt eliberate). După cum

am precizat anterior, la crearea unui obiect este absolut necesar să se precizeze modelul (clasa) după care este „turnat", dar, pentru ca el să poată fi folosit, variabilele, al căror tip şi număr au fost precizate în definiția clasei, ar trebui să primească valori. Această operațiune de asignare de valori variabilelor obiectului în momentul naşterii sale sunt efectuate de nişte metode speciale, numite *constructori*.

Într-o clasă, se pot defini mai mulți constructori, alegerea celui potrivit făcându-se în funcție de parametrii ceruți la instanțiere (la crearea unui obiect din clasa respectivă). Chiar dacă în definiția unei clase nu este implementat în mod explicit un constructor, sistemul generează un constructor implicit fără nici un parametru.

Existența constructorilor poate scuti programatorii de grija apelării în mod explicit a unor metode care să aloce resurse obiectului. Faptul că starea obiectului în momentul naşterii sale depinde de parametrii precizați (constructorul fiind ales în funcție de tipul şi numărul acestora) conferă o mare flexibilitate codului, mai ales dacă se ia în considerare că metodele de tip constructor se pot moşteni.

La distrugerea obiectelor este necesară eliberarea resurselor alocate, acțiune ce revine metodelor de tip *destructor*. *Exemplu*: un obiect, pe parcursul „vieții" sale, alocă dinamic spațiu de memorie; dacă în destructor spațiul de memorie respectiv nu este eliberat, atunci acesta nu va mai putea fi folosit de către alt proces. Acest tip de eroare este foarte periculos, putând duce la „sufocarea" (*memory leak*) sistemului din lipsă de resurse (pentru a evita situații de acest gen, creatorii limbajului de programare Java au eliminat posibilitatea alocării dinamice a memoriei prin pointeri şi au introdus un mecanism automat de „colectare a gunoiului" – *garbage collection* –, prin care sunt eliberate resursele ce au fost folosite de obiectele distruse).

O altă sarcină foarte importantă pe care destructorul trebuie să o îndeplinească este asigurarea că dispariția sa nu va crea nici o încurcătură. *Exemplu*: dacă se doreşte ştergerea unui obiect aflat într-o listă de obiecte, atunci este necesară „avertizarea" obiectelor vecine de la stânga şi de la dreapta, pentru ca ele să stabilească „legături directe", altfel, lista s-ar rupe în două.

Elemente statice ale obiectelor

Pe lângă variabilele şi metodele declarate în mod „normal", există posibilitatea declarării lor *statice*. Dacă o variabilă este declarată statică în definiția unei clase, atunci compilatorul va crea doar o singură copie a ei, indiferent de numărul obiectelor instanțiate din clasa respectivă. Toate aceste obiecte vor face referire practic la o singură variabilă, deci valoarea stocată de ea va fi aceeaşi pentru toate obiectele, spre deosebire de „elementele normale", pentru care există câte o copie aferentă fiecărui obiect. *Exemplu*: variabila „cont" în clasa Object din programul problemei 1 (problema fracționară a rucsacului) din capitolul *Greedy*.

Restricții:

- O metodă statică poate folosi doar elemente statice din clasa în cauză.
- Nu se pot supraîncărca metode statice cu metode non-statice şi nici invers.

Elementele statice sunt folosite mai ales pentru controlul la nivel de clasă al programelor. Cu ajutorul lor se poate controla comportamenul tuturor obiectelor instanțiate dintr-o anumită clasă.

Avantajele utilizării programării orientate obiect

Ușurința proiectării şi reutilizării codului

Odată ce este testată corectitudinea funcționării unor obiecte dintr-o aplicație, acestea vor putea fi folosite fără nici o problemă şi în altă aplicație. Acest avantaj poate fi valorificat prin constituirea de biblioteci de obiecte. În ceea ce priveşte proiectarea, se facilitează descompunerea problemelor complexe în subprobleme simple, care pot fi uşor modelate cu ajutorul obiectelor (variabilele vor descrie proprietățile obiectelor modelate şi metodele acțiunilor lor).

Abstractizare

Proiectanții pot obține o imagine de ansamblu, urmărind comportarea obiectelor şi interacțiunile dintre ele, detaliile fiind înglobate în compoziția obiectelor.

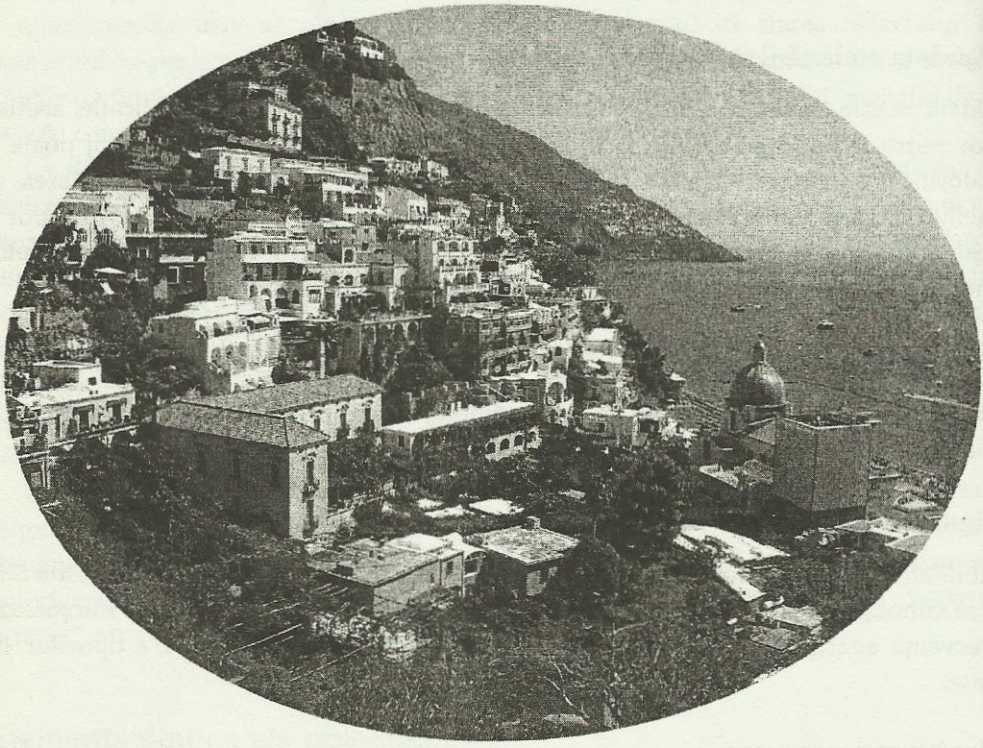
Siguranța datelor

Abilitatea obiectelor de a se comporta ca nişte „cutii negre", de a putea fi folosite fără a se cunoaşte compoziția lor, asigură confidențialitatea datelor utilizate şi micşorează frecvența aparițiilor şi efectul erorilor legate de manipularea greşită a tipurilor de date.

Probleme propuse

1. Ce este *abstractizarea datelor*? Selectați trei exemple din cartea de față.
2. Care este diferența dintre un obiect şi o clasă? Se poate crea un obiect fără a crea clasa după care este instanțiat?
3. Care este diferența dintre o variabilă locală şi o variabilă membru a unei clase?
4. Care este diferența dintre un câmp static şi unul nestatic?
5. Cum se pot ascunde datele în C++?
6. Ce este *polimorfismul*? Găsiți exemple de polimorfism în programele cărții de față.

7. Care este mecanismul de creare şi distrugere a obiectelor în C++? La ce poate conduce păstrarea obiectelor care nu mai sunt necesare dintr-un anumit punct al aplicației?
8. Ce este *moştenirea*? Găsiți trei exemple sugestive şi implementați programe simple pentru a ilustra conceptul.
9. Enumerați avantajele utilizării programării orientate obiect.
10. Explicați conceptul de *încapsulare*. Dați trei exemple.



Positano, Italia

Capitolul 3
Introducere în *STL*

10 probleme complet rezolvate, 16 probleme propuse

- Istoric, concepte de bază
- Programarea generică
- Containere
- Iteratori
- Algoritmi
- Functori
- Încă şapte probleme rezolvate
- Întrebări, exerciţii, probleme propuse

Possibly one of the most boring terms in history,
for one of the most exciting tools in history.
Scott Field

Istoric, concepte de bază

Standard Template Library (STL) este o parte foarte importantă a limbajului C++, devenită standard al limbajului încă din 1999. Ea furnizează diverse implementări ale unor structuri şi algoritmi des utilizate, cum ar fi containere, iteratori, algoritmi, functori şi alocatori. Este o bibliotecă generică, aceasta însemnând că aproape toate componentele sale sunt parametrizate: fiecare componentă este în *STL* un şablon (tip generic, *template*). Este indicat să fie utilizată biblioteca *STL* oricând este posibil, deoarece este foarte practică, eficientă şi robustă.

Programarea generică

Programarea generică este o paradigmă de programare ce permite implementarea unor clase şi metode, astfel încât să poată fi folosite cu diverse tipuri de date. Diferite limbaje de programare suportă conceptul de tipuri generice (şabloane, *template*-uri), astfel încât tipul variabilei nu este stabilit, ci generic (un fel de polimorfie generalizată sau polimorfie parametrizată), dar trebuie să satisfacă o serie de condiţii. Astfel de limbaje poartă numele de *limbaje generice*. În *header*-ul de algoritmi al *STL* se găsesc numeroşi algoritmi de bază, des utilizaţi în implementarea aplicaţiilor, care sunt cât mai independenţi posibil de tipurile de date cu care lucrează.

Ca exemplu, scriem un program care implementează o funcţie de interschimbare a două elemente. Deoarece tipul este generic, vom putea folosi aceasta funcţie pentru orice tipuri de variabile. În programul principal se testează pentru şiruri de caractere (*std::string*), tipul *float* şi vectori.

Problema 1. Scrieţi un program care implementează o funcţie generică pentru interschimbarea a două elemente şi o testează pentru tipuri diferite de argumente.

Program 1

```
#include <string>
#include <vector>
#include <iostream>

using namespace std;

template <class T> void interschimba(T &x, T &y){
  T aux;
  aux = x;
    x = y;
    y = aux;
};

int main(){

  string s1 = " ziua!";
  string s2 = "Buna";
  interschimba(s1, s2);
  cout << s1 << s2 << endl
       << endl;

  float f1, f2;
  cout << "f1 = "; cin >> f1;
  cout << "f2 = "; cin >> f2;
  cout << "--------" << endl;
  interschimba(f1, f2);
  cout << "f1 = " << f1 << endl
       << "f2 = " << f2 << endl
       << endl;

  vector<int> v1;
  vector<int> v2;
  int i;
  for (i=0; i<10; i++) v1.push_back(i);
  for (i=100; i>50; i-=10) v2.push_back(i);
  interschimba(v1, v2);
  cout << "v1: ";
  for (i=0; i<(int)v1.size(); i++) cout << v1[i] << " ";
  cout << endl << "v2: ";
  for (i=0; i<(int)v2.size(); i++) cout << v2[i] << " ";
}
```

Efectul programului

```
Buna ziua!

f1 = 4.53
f2 = 2.09
--------
f1 = 2.09
f2 = 4.53

v1: 100 90 80 70 60
v2: 0 1 2 3 4 5 6 7 8 9
```

Observăm astfel că perechile de variabile au fost inversate folosind aceeaşi funcţie *interschimba()*, dar cu parametri de tipuri diferite (specializare explicită). Acest lucru este posibil datorită tipului generic *T*, folosit în implementarea metodei.

Containere

Containerele *STL* (colecţii) implementează multe modalităţi de a stoca datele, împreună cu o serie de metode uzuale pentru modificarea şi exploatarea acestora. Ele pot stoca orice tip de elemente şi cele mai multe dintre ele sunt generalizate ca *template*-uri. Foarte des utilizate sunt containerele *vector*, *string* şi *map*.

Containerele secvenţiale conţin secvenţe de lungime variabilă de elemente de acelaşi tip *T*:

- *<vector>*: tablou dinamic cu elemente de un tip dat *T*, acces rapid la orice element;
- *<deque>*: de asemenea, un tablou dinamic de elemente, care aduce în plus inserarea şi ştergerea rapide ale unui element de la începutul, respectiv sfârşitul secvenţei;
- *<list>*: este implementată ca o listă dublu înlănţuită, inserarea şi eliminarea unui element la ambele capete ale secvenţei se execută foarte rapid.

Containerele asociative conţin perechi cheie/valoare, în care accesul la valori se face prin intermediul cheilor. Elementele sunt sortate după cheie şi sunt implementate ca arbori binari echilibraţi:

- *<map>*: furnizează accesul la valori folosind orice tip de cheie. Elementele sunt de tipul *pair<const Key, T>*, unde primul element este cheia sortată, iar al doilea este valoarea asociată ei. Pentru fiecare cheie există cel mult o valoare. Tipul (clasa) pentru cheie trebuie să suporte operaţia de comparaţie „mai mic" pentru a putea executa operaţia de sortare;

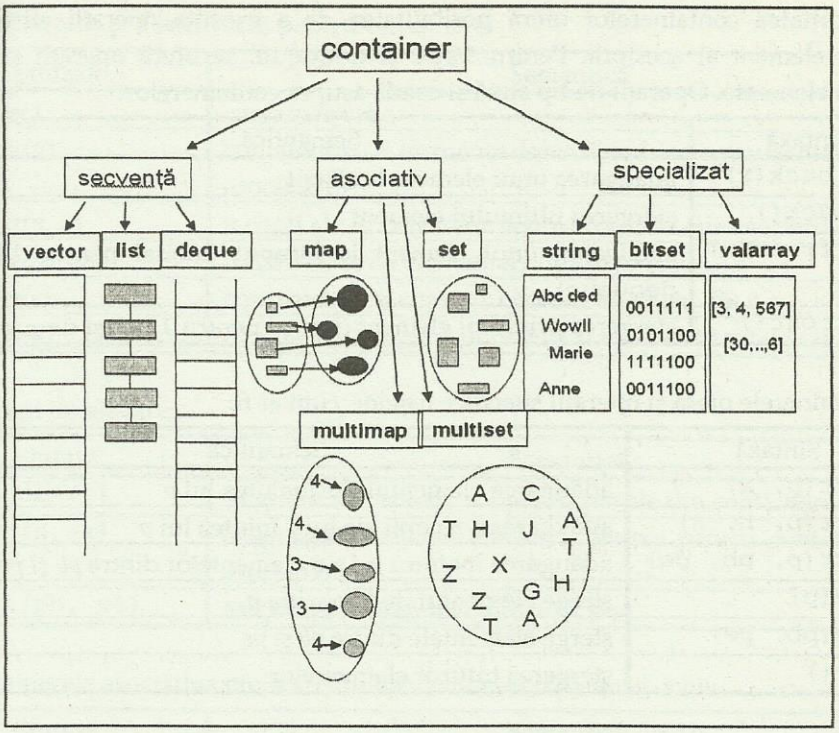
- *multimap*: este același lucru ca și *map*, numai că este permisă apariția repetată a aceleiași chei. Declarația clasei se găsește în *<map>*;
- *<set>*: furnizează o mulțime ordonată de elemente (fiecare element apare exact o dată), nu implementează (așa cum ar fi de așteptat) și operațiile uzuale pe mulțimi (reuniune, intersecție etc.);
- *multiset*: multimulțime ordonată de elemente (un element poate să apară repetat). Declarația clasei se află în *<set>;*

Containerele adaptor sunt construite pe baza altor containere și sunt folosite doar pentru a forța regulile de acces. Tocmai din cauza acestor restricții de acces, containerele adaptor nu suportă iteratori:

- *<stack>*: structură de tip stivă, oferă acces *LIFO* (*Last In First Out* – ultimul adăugat este primul extras);
- *<queue>*: structură de tip coadă, oferă acces *FIFO* (*First In First Out* – primul adăugat este primul extras);
- *priority_queue*: coadă în care elementele au priorități, care determină și ordinea de acces la ele. Declarația sa se găsește în *<queue>*.

Containerele specializate conțin elemente specializate; de exemplu, tipuri de date specifice, rutine utilitare, implementări limitate (dar eficiente). *Stroustrup* numește aceste containere „aproape containere", datorită faptului că fiecare dintre ele are o serie de limitări comparativ cu containerele standard:

- *<string>*: oferă multe funcții utile și eficiente pentru lucrul cu șiruri de caractere;
- *<bitset>*: un *bitset<N>* este un câmp cu *N* biți. El se diferențiază de *vector<bool>* prin faptul că are o lungime fixă. Pentru mulțimi mici de biți, C++ oferă posibilitatea folosirii operatorilor pe biți pentru numere întregi (*&* – și, *|* – sau, *^* – sau exclusiv, *<<* – deplasare spre stânga, *>>* – deplasare spre dreapta, *~* – negație). Clasa *bitset<N>* generalizează această abordare și oferă abordarea comodă a mulțimilor de *N* biți indexate de la 0 la *N* – 1, unde *N* trebuie să fie cunoscut la compilare;
- *<valarray>*: este un vector optimizat pentru operații numerice, dar nu are o comportare de container standard.



Containere (colecții de date) uzuale în STL

Timpul necesar pentru operații uzuale asupra containerelor secvențiale este redat în tabelul de mai jos.

Operație	`vector`	`dequeue`	`list`
Acces la primul element	constant	constant	constant
Acces la ultimul element	constant	constant	constant
Acces la un element aleator	constant	constant	liniar
Adăugare/ştergere la început	liniar	constant	constant
Adăugare/ştergere la sfârşit	constant	constant	constant
Adăugare/ştergere aleatoare	liniar	liniar	constant

Unele elemente pot fi accesate direct:

Sintaxă	Semantică
`front()`	primul element
`back()`	ultimul element
`[]`	acces fără verificare la un index dat (nu şi pentru liste)
`at()`	acces verificat la un index dat (nu şi pentru liste)

Majoritatea containerelor oferă posibilitatea de a executa operații eficiente cu ultimul element al acestora. Pentru `list` și `dequeue`, se oferă operații și pentru primele elemente. Operații de tip stivă și coadă asupra containerelor:

Sintaxă	Semantică
`push_back(x)`	adăugarea unui element la sfârșit
`pop_back()`	ștergerea ultimului element
`push_front(x)`	adăugarea unui element la început (numai pentru `list` și `dequeue`)
`pop_front()`	ștergerea primului element (numai pentru `list` și `dequeue`)

Containerele oferă și operații specifice listelor, cum ar fi:

Sintaxă	Semantică
`insert(p, x)`	adăugarea elementului *x* înaintea lui *p*
`insert(p, n, x)`	adăugarea a *n* copii ale lui *x* înintea lui *p*
`insert(p, pb, pe)`	adăugarea înaintea lui *p* a elementelor dintre *pb* și *pe*
`erase(p)`	șterge elementul de la poziția *p*
`erase(pb, pe)`	șterge elementele dintre *pb* și *pe*
`clear()`	ștergerea tuturor elementelor

Toate containerele oferă alte operații uzuale:

Sintaxă	Semantică
`size()`	returnează numărul elementelor
`empty()`	testează dacă are elemente
`max_size()`	dimensiunea celui mai mare container posibil
`capacity()`	spațiul rezervat (numai pentru `vector`)
`reserve()`	rezervarea spațiului (numai pentru `vector`)
`resize()`	schimbarea numărului de elemente (numai `vector`, `deque`, `list`)
`swap()`	interschimbarea elementelor a două containere
`==`	compară dacă două containere au același conținut
`!=`	compară dacă două containere au conținuturi diferite
`<`	compararea ordinii lexicografice

Constructorii şi destructorii pentru containere:

Sintaxă	Semantică
container()	container vid
container(*n*)	container cu *n* elemente *default* (exclus pentru containerele asociative)
container(*n*, *x*)	*n* copii ale lui *x* (exclus pentru containerele asociative)
container(*pb*, *pe*)	iniţializarea cu elementele dintre *pb* şi *pe*
container(*c*)	constructor de copiere; copie elementele din *c*
~*container*()	destructor; şterge containerul şi toate elementele sale

Operaţii de alocare:

Sintaxă	Semantică
`operator=(c)`	asignare de copiere; copie elementele din containerul *c*
`assign(n, x)`	asignarea a *n* copii ale lui *x* (exclus pentru containerele asociative)
`assign(pb, pe)`	asignează elementele dintre *pb* şi *pe*

Containerele asociative oferă accesul la elemente cu ajutorul cheii:

Sintaxă	Semantică
`operator[](k)`	accesul la elementul cu cheia *k* (pentru containerele cu cheie unică)
`find(k)`	căutarea elementului cu cheia *k*
`lower_bound(k)`	căutarea primului element cu cheia *k*
`upper_bound(k)`	căutarea primului element cu cheie mai mare decât *k*
`equal_range(k)`	găseşte `lower_bound()` şi `upper_bound()` pentru *k*

Iteratori

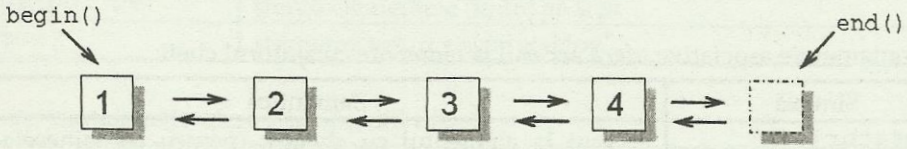
Iteratorii sunt folosiţi pentru a naviga între elementele unui container, fără a fi necesar să ştim tipul elementelor acestuia. El este asociat la declarare unui anumit tip de container. Un iterator poate fi incrementat, decrementat, dereferenţiat, comparat cu un altul.

Iteratorii sunt abstractizări generalizate ale pointerilor, văzuţi deci ca nişte „cutii negre", folosite de cele mai multe ori în practică pentru a aplica algoritmii asupra containerelor. Dacă se utilizează un telefon pentru a suna pe cineva în altă ţară, de exemplu, nu este necesară întreaga informaţie despre sistemul telefonului-destinaţie. Este necesar însă ca acesta să suporte o serie de operaţii de bază, cum ar fi: selectarea numărului, soneria, raportarea stării, închiderea conexiunii la aşezarea receptorului,

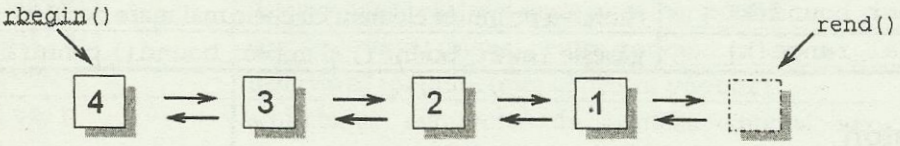
care sunt necesare pentru stabilirea comunicării. În aceeași manieră, dacă o clasă suportă un minimum de tipuri de iteratori pentru un anumit algoritm, atunci algoritmul va funcționa pentru respectivul container.

Condiția care se folosește de obicei pentru terminarea unei bucle este puțin diferită de cea normală. În mod uzual, se folosește de multe ori condiția „mai mic sau egal". În cazul iteratorilor, se pornește, de obicei, de la `begin()` și condiția de oprire este înainte de egalitatea cu `end()`. Este important de știut că `end()` returnează un iterator care pointează la un element imediat după ultimul. Acesta nu se poate dereferenția, dar se folosește pentru teste de egalitate cu alți iteratori de același tip. Operatorul de preincrementare `++` poate conduce uneori la performanțe superioare operatorului de postincrementare, deoarece nu creează o copie temporară a valorii precedente.

Perechea *begin()/end()* furnizează elementele containerului în ordinea directă folosind *iterator* (elementul 0, elementul 1, elementul 2 ș.a.m.d.). Perechea *rbegin()/rend()* furnizează elementele containerului în ordine inversă folosind un *reverse_iterator* (elementul *n*–1, elementul *n*–2, elementul *n*–3 ș.a.m.d.). O secvență parcursă cu *iterator* poate fi astfel reprezentată:



Cu *reverse_iterator* se poate reprezenta astfel:



Programul pentru problema următoare este sugestiv privind utilizarea celor două variante de iteratori.

Problema 2. Scrieți un program care citește de la tastatură un vector de numere reale și apoi îl afișează folosind *iterator* și *reverse_iterator*.

Program 2

```
#include <iostream>
#include <vector>

using namespace std;
int main(){
  vector<short> v;
  short n, t;
  cout << "n = ";
  cin >> n;
  cout<< "Elementele: ";
  for (short i=0; i<n; i++){
    cin >> t;
    v.push_back(t);
  }
  vector<short>::iterator it;
  cout << "\nParcurgere directa: ";
  for(it=v.begin(); it<v.end(); ++it){
    cout << *it << " ";
  }

  vector<short>::reverse_iterator rIt;
  cout <<"\nParcurgere inversa: ";
  for(rIt=v.rbegin(); rIt<v.rend(); ++rIt){
    cout << *rIt << " ";
  }
}
```

Efectul programului:

```
n = 5
Elementele: 34 -23 4 67 1

Parcurgere directa: 34 -23 4 67 1
Parcurgere inversa: 1 67 4 -23 34
```

Algoritmii din *header*-ul *<algorithm>*, cum ar fi, de exemplu, *sort* sau *random_shuffle*, nu cunosc tipul containerelor, dar folosesc anumiți iteratori, ce trebuie să fie suportați de acestea. Iteratorii pot fi priviți şi ca o interfață dintre containere şi algoritmi. Există cinci categorii de iteratori:

- iterator cu acces uniform: *Random_Access_Iterator*;
- iterator bidirecțional: *Bidirectional_Iterator*;
- iterator de înaintare: *Forward_Iterator*;

- iterator de intrare: *Input_Iterator*;
- iterator de ieşire: *Output_Iterator*.

Pentru toate variantele de iteratori, există şi variante *const* (*const_iterator*, *const_reverse_iterator*), care se referă la iteratori ce nu modifică starea elementelor referențiate. Aceştia sunt recomandați în cazul în care nu este necesară modificarea conținutului elementelor.

Algoritmi

Algoritmii *STL* sunt funcții generice C++, care execută operații uzuale asupra containerelor. Pentru a putea fi capabili să opereze cu diferite tipuri de containere, ei nu primesc ca parametri containerele, ci iteratori care specifică o parte a lor. Unii algoritmi au nevoie doar de suport pentru intrare (*Input_Iterator*), pe când alții au nevoie de acces uniform (*Random_Access_Iterator*, de exemplu, pentru *sort*()).

Utilizarea algoritmilor pare la prima vedere dificilă, dar, odată folosiți, devine foarte practică. Mulți dintre aceştia au interfețe foarte asemănătoare, ce uşurează înțelegerea lor. În bilbioteca *<algorithm>* se găsesc 60 de algoritmi care includ operații de sortare (*sort, merge, min, max* etc.), căutare (*find, count, equal* etc.), operații de mutare (*transform, replace, fill, rotate, shuffle* etc.) şi operații numerice generalizate (*acumulate, adjancent difference* etc.). Ei se găsesc în spațiul de nume *std* şi sunt declarați în *<algorithm>*.

Operațiile care nu modifică secvențele sunt cele care extrag informații despre un container; de exemplu, poziția unui element în container sau numărul de elemente:

Sintaxă	Semantică
`for_each()`	pentru fiecare element din intervalul dat execută
`find()`	prima apariție a unei valori
`find_if()`	primul element care satisface un predicat
`find_first_of()`	element al unei secvențe în altă secvență
`adjacent_find()`	găsirea unei perechi de elemente
`count()`	numărul de elemente cu o anumită valoare
`count_if()`	numărul de elemente care satisfac un predicat
`mismatch()`	primele elemente diferite a două secvențe
`equal()`	testează dacă două secvențe conțin aceleaşi valori
`search()`	prima apariție a unei subsecvențe
`find_end()`	ultima apariție a unui element
`search_n()`	prima subsecvență cu *n* elemente egale

Algoritmii care modifică secvențele:

Sintaxă	Semantică
`transform()`	pentru fiecare element din intervalul dat execută operație
`copy()`	copierea tuturor elementelor începând cu primul
`copy_backwards()`	copierea tuturor elementelor începând cu ultimul
`swap()`	interschimbarea a două elemente
`iter_swap()`	interschimbarea a două elemente indicate de doi iteratori
`swap_ranges()`	interschimbarea tuturor elementelor a două secvențe
`replace()`	înlocuirea unor elemente cu o altă valoare
`replace_if()`	înlocuirea unor elemente care satisfac un predicat
`replace_copy()`	copierea cu înlocuirea unei valori date
`replace_copy_if()`	copierea cu înlocuirea unor elemente care satisfac un predicat
`fill()`	înlocuirea tuturor elementelor cu o valoare dată
`fill_n()`	înlocuirea a *n* elemente cu o valoare dată
`generate()`	înlocuirea tuturor elementelor cu rezultatul unei operații
`generate_n()`	înlocuirea a *n* elemente cu rezultatul unei operații
`remove()`	ştergerea elementelor cu o anumită valoare
`remove_if()`	ştergerea elementelor care satisfac un predicat
`remove_copy()`	copierea elementelor cu eliminarea unei valori
`remove_copy_if()`	copierea cu eliminarea elementelor ce satisfac un predicat
`unique()`	eliminarea duplicatelor consecutive
`unique_copy()`	copierea elementelor cu eliminarea duplicatelor consecutive
`reverse()`	inversarea elementelor secvenței
`reverse_copy()`	copierea elementelor în ordine inversă
`rotate()`	rotirea elementelor
`rotate_copy()`	copierea elementelor cu rotirea lor
`random_shuffle()`	amestecarea elementelor

STL oferă şi foarte mulți algoritmi pentru sortarea, căutarea şi manipularea secvențelor sortate:

Sintaxă	Semantică
`sort()`	sortarea secvenței
`stable_sort()`	sortarea (aceleaşi elemente îşi păstrează ordinea)
`partial_sort()`	sortarea primei părți a secvenței
`partial_sort_copy()`	copierea şi apoi sortarea primei părți

`nth_element()`	poziționarea celui de-al *n*-lea element pe poziția finală în secvența sortată
`lower_bound()`	găsirea primului element în secvența sortată
`upper_bound()`	ultimul element în secvența sortată
`equal_range()`	găsirea unei subsecvențe cu o anumită valoare
`binary_search()`	găsirea unei valori într-o secvență sortată
`merge()`	combinarea a două secvențe sortate
`inplace_merge()`	combinarea a două secvențe sortate consecutive
`partition()`	plasarea în față a elementelor care satisfac un predicat
`stable_partition()`	plasarea în față a elementelor care satisfac un predicat, cu păstrarea ordinii relative

Algoritmii pentru mulțimi:

Sintaxă	Semantică
`includes()`	verifică incluziunea unei secvențe în alta
`set_union()`	reuniunea a două secvențe
`set_intersection()`	intersecția a două secvențe
`set_difference()`	mulțime sortată cu elementele care sunt în prima secvență și nu sunt în a doua
`set_symmetric_difference()`	mulțime sortată cu elementele care se află exact în una dintre cele două mulțimi date

Operațiile *heap* păstrează o secvență într-o stare astfel încât rămâne simplu de sortat, atunci când este nevoie:

Sintaxă	Semantică
`make_heap()`	pregătirea unei secvențe pentru manipularea ca *heap*
`push_heap()`	introducerea unui element în *heap*
`pop_heap()`	eliminarea unui element din *heap*
`sort_heap()`	sortarea *heap*-ului

Algoritmii pentru selectarea anumitor elemente pe baza unor comparații:

Sintaxă	Semantică
`min()`	returnează elementul mai mic dintre două valori
`max()`	returnează cea mai mare dintre două valori
`min_element()`	returnează cel mai mic element dintr-o secvență

`max_element()`	returnează cel mai mare element dintr-o secvență
`lexicographical_compare()`	compararea lexicografică a două secvențe

STL oferă următoarele metode pentru operațiile cu permutări:

Sintaxă	Semantică
`next_permutation()`	următoarea permutare în ordine lexicografică
`prev_permutation()`	permutarea precedentă în ordine lexicografică

Functori

Un obiect de tip funcție (*functor*) este un obiect care poate fi apelat ca și funcție. O funcție obișnuită este un astfel de obiect; la fel este un obiect ce are definit sau supraîncărcat operatorul de apel de funcție (*operator*()). Aceste obiecte de tip funcție au importanță deosebită pentru *STL*.

Conceptele de bază pentru obiecte de tip funcție sunt *generator*, *funcție unară* și *funcție binară*: acestea descriu, respectiv, dacă obiectele pot fi apelate ca *f*(), *f*(*x*) sau *f*(*x*, *y*). Această listă poate fi extinsă la *funcție ternară* etc., dar, în practică, toți algoritmii din *STL* folosesc functori cu cel mult două argumente. Toate conceptele de obiecte de tip funcție folosite în *STL* sunt rafinări ale acestora trei.

Obiectele de tip funcție care returnează *bool* sunt un caz special. O funcție unară ce returnează *bool* se numește *predicat* (*unar*), o funcție binară ce returnează *bool* se numește *predicat binar*.

Problema 3. Scrieți un program care găsește poziția primului element prim într-un vector de numere naturale, folosind două variante: căutarea secvențială în vector și metoda *find_if* din *header*-ul *algorithm* cu al treilea parametru predicat binar (funcție de testare a primalității).

Program 3

```
#include <vector>
#include <iostream>
#include <algorithm>

using namespace std;

bool prim(unsigned n){
  if(n>2 && n%2==0) return false;
  for(unsigned i=3; i*i<=n; i+=2)
    if(n%i==0) return false;
```

```
  return true;
}

int main(){
  vector<unsigned> v;
  int n, t;
  cout << "n = "; cin >> n;
  cout << "Elementele: ";
  while(n--){
    std::cin >> t;
    v.push_back(t);
  }

  // prima variantă: căutare iterativă în vector
  vector<unsigned>::iterator it;
  bool found = false;
  for(it=v.begin(); !found && it!=v.end(); it++)
    if(prim(*it)) found = true;
  if(found){
    cout << "Primul element prim la pozitia "
         << it-v.begin();
  }else{
    cout << "Tabloul nu contine elemente prime!";
  }

  //a doua variantă: prim este predicat binar pentru algoritmul find_if
  it = find_if(v.begin(), v.end(), prim);
  if(it!=v.end()){
    cout << "\nPrimul element prim la pozitia "
         << it - v.begin() + 1;
    }else{
     cout << "\nTabloul nu contine elemente prime!";
  }

}
```

Efectul programului:

```
n = 5
Elementele: 45 6 78 37 56
Primul element prim la pozitia 4
Primul element prim la pozitia 4
```

Prima variantă folosește un iterator, care se deplasează de la stânga spre dreapta în vector până când elementul curent este prim sau se ajunge la sfârșitul vectorului. A doua variantă utilizează algoritmul *find_if* din *header*-ul <*algorithm*>: returnează primul

element pentru care predicatul binar este satisfăcut; dacă nu există un astfel de predicat, va returna *v.end()*.

Încă şapte probleme rezolvate

Problema 4. Scrieţi un program care alege aleator o variantă pentru LOTO 6/49.

Program

```
#include <vector>
#include <iostream>
#include <algorithm>
#include <iterator>

const short NMAX = 49;
const short M = 6;

int main(){
  std::vector<short> v;
  for(short i=0; i<NMAX; i++)
    v.push_back(i+1);
  random_shuffle(v.begin(), v.end());
  copy(v.begin(), v.begin()+M,
    std::ostream_iterator<short> (std::cout, " "));
  return 0;
}
```

Efectul programului

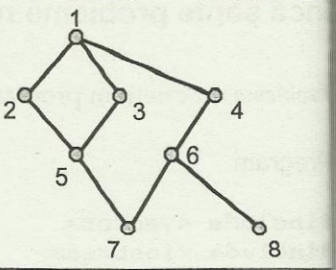
```
12 1 9 44 14 27
```

Vectorul *v* este completat dinamic cu numerele 1, 2, ..., *NMAX*, apoi este amestecat folosind metoda *random_shuffle()*, după care se vor afişa primele *M* elemente ale sale (presupunem că $M \leq NMAX$). Pentru scrierea în fluxul de ieşire (pe ecran – *std::cout*), folosim metoda *copy()*, apelul din program este echivalent şi cu:

```
for(short i=0; i<M; i++)
   std::cout << v[i] << "  ";
```

Ca exerciţiu, modificaţi programul astfel încât afişarea să se facă folosind iteratori (variantele *iterator* şi *reverse_iterator*) şi folosind metoda *at()* (acces verificat) în locul metodei *copy*.

Problema 5. Scrieți un program care afișează parcurgerea *DFS (depth first search)* a unui graf dat prin matricea de adiacență. Exemplu:

graph.in	dfs.out	
8 1 0 1 1 1 0 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 0 0 0 1 1 0 0 0 1 0 0 0 0 1 0 0 1 1 0 0 0 0 1 1 0 0 0 0 0 0 0 1 0 0	1 2 5 3 7 6 4 8	

Acest algoritm traversează graful în adâncime. Începând cu vârful inițial, se va vizita primul vecin al acestuia încă neprocesat, după aceea primul vecin neprocesat al acestuia ş.a.m.d. Atunci când un nod nu mai are vecini neprocesați, se va trece la predecesorul acestuia, pentru care se caută vecinii neprocesați. Algoritmul foloseşte o stivă (engl. *stack*):

```
ALGORITM_DFS(Graf G, Vârf y)
  Inițializează stiva S cu {y}
  While (S NOT Null) Do
    k ← S.top() // elemenul din vârful lui S, fără ştergerea acestuia
    Marchează$iProcesează k
    j ← primul vecin nevizitat al lui k
    If (j există) Then
      Marchează$iProcesează j
      S.push(j)
    End_If
    Else S.pop() End_Else      // şterge elementul din vârful lui S
  End_While
ALGORITM_DFS(Graf G, Vârf y)
```

Modelăm algoritmul de mai sus folosind containerul adaptor *std::stack*.

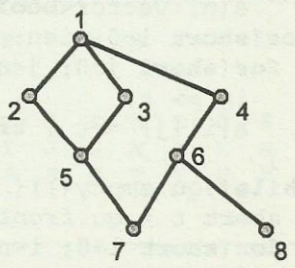
Program 5

```
#include <stack>
#include <fstream>
#include <vector>
#include <algorithm>
```

```
int main(){
  std::ifstream in("graph.in");
  std::ofstream out("dfs.out");
  short n, k, a[100][100];
  std::stack<short> st;
  std::vector<short> v;
  in >> n >> k;
  st.push(k-1);
  for(short i=0; i<n; i++)
    for(short j=0; j<n; j++)
      in >> a[i][j];
  while(!st.empty()){
    short t = st.top();
    for(short i=n; i>=0; --i)
      if(a[t][i] &&
        find(v.begin(), v.end(), i)==v.end()
        ){
          st.push(i);
          }
      if(find(v.begin(), v.end(), t)==v.end()){
        v.push_back(t);
        } else st.pop();
    }
  for(short i=0; i<v.size(); i++)
  out << v[i]+1 << " ";
  return 0;
```

Observăm utilizarea metodelor *push, pop* şi *top* specifice stivei şi a algoritmului find() din *header*-ul *<algorithm>*.

Problema 6. Scrieţi un program care afişează parcurgerea *BFS (breadth first search)* a unui graf dat prin matricea de adiacenţă. Exemplu:

graph.in	bfs.out	
8 1 0 1 1 1 0 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 0 0 0 1 1 0 0 0 1 0 0 0 0 1 0 0 1 1 0 0 0 0 1 1 0 0 0 0 0 0 0 1 0 0	1 2 3 4 5 6 7 8	

Acest algoritm traversează graful în lățime. Începând cu vârful inițial, se vor vizita apoi toți vecinii nevizitați ai acestuia, după aceea toți vecinii nevizitați ai fiecăruia dintre ei ș.a.m.d. Algoritmul folosește o coadă (engl. *queue*):

```
ALGORITM_BFS(Graf G, Vârf y)
  Inițializează coada Q cu {y}
  While (Q NOT Null) Do
    k ← Q.pop()              // returnarea și ștergerea primului element din Q
    If (k not marked) Then
      MarkAndProcess k
      Q.push(all not marked Successors from k)
    End_If
  End_While
ALGORITM_BFS(Graf G, Vârf y)
```

Modelăm algoritmul de mai sus folosind containerul adaptor *std::queue*.

Program 6

```
#include <queue>
#include <fstream>
#include <vector>
#include <algorithm>

using namespace std;
int main(){
  std::ifstream in("graph.in");
  std::ofstream out("bfs.out");
  std::queue<short> qu;
  std::vector<short> v;
  short n, k, t;
  in >> n >> k;
  qu.push(k-1);
  std::vector<std::vector<bool> >
      a(n, vector<bool>(n, true));
  for(short i=0; i<n; i++)
    for(short j=0; j<n; j++){
      in >> t;
      a[i][j] = t ? true : false;
    }
  while(!qu.empty()){
    short t = qu.front();
    for(short i=0; i<n; ++i)
      if(a[t][i] &&
        find(v.begin(), v.end(), i)==v.end()){
```

```
            qu.push(i);
        }
    if(find(v.begin(), v.end(), t)==v.end()){
      v.push_back(t);
    } else qu.pop();
}
for(size_t i=0; i<v.size(); i++)
out << v[i]+1 << " ";
return 0;
}
```

Observăm că diferențele față de programul anterior nu sunt mari. Prima dintre ele este includerea *<queue>* în locul *<stack>*. Metodele *push*, *pop* şi *empty* au aceeaşi signatură şi semnificație ca şi la *stack*. Metoda pentru extragerea unui element din coadă este *front* (la stivă este *top*). Ca variație şi pentru a prezenta o nouă modalitate de a declara dinamic un tablou bidimensional, matricea *a* este considerată a fi „vector de vectori cu elemente de tip *bool*", care, la declarare, este şi inițializată cu $n \times n$ elemente *true*. Pentru a reda prima parcurgere *BFS* în ordine lexicografică, folosim în instrucțiunea *for* evidențiată căutarea vecinilor de la 0 la $n-1$. *Notă*: rezolvări C ale problemelor 5 şi 6 se găsesc de exemplu în [Log06].

Problema 7. Determinați mulțimea şi multimulțimea literelor dintr-un citat dat, fără să se facă distincție între litere mari şi mici. Exemplu:

citat.in	**multime.out**
Problema e ca daca nu risti, risti chiar mai mult.	Multime: A B C D E H I L M N O P R S T U Multimultime: A A A A A A B C C C D E E H I I I I I I L L M M M N O P R R R R S S T T T U U
Cel ce nu stie incotro vrea, sa nu se mire ca nu ajunge nicaieri.	Multime: A C E G I J L M N O R S T U V Multimultime: A A A A A C C C C C E E E E E E E E G I I I I I I J L M N N N N N N O O R R R R S S S T T U U U U V

Vom folosi containerele asociative *std::set* (mulțime) și *std::multiset* (multimulțime). Toate literele citite se vor introduce în două containere și apoi le vom afișa în fișierul de ieșire folosind iteratori constanți (aceștia se recomandă pentru că elementele referențiate de ele nu își schimbă valorile). Programul este sugestiv în acest sens.

Program 7

```
#include <set>
#include <fstream>

int main(){
  std::ifstream in("citat.in");
  std::ofstream out("multimi.out");

  std::set<char> s;
  std::set<char>::const_iterator sIt;
  std::multiset<char> m;
  std::multiset<char>::const_iterator mIt;
  char c;

  while(in && !in.eof() && in>>c){
    if(isalpha(c)){
      s.insert(toupper(c));
      m.insert(toupper(c));
    }
    }

  out << "Multime: " << std::endl;
  for(sIt=s.begin(); sIt != s.end(); sIt++)
  out << *sIt << "  ";

  out << "\n\nMultimultime: " << std::endl;
  for(mIt=m.begin(); mIt != m.end(); mIt++)
  out << *mIt << "  ";

  return 0;
}
```

Problema 8. Scrieți un program care pentru un număr natural dat n ($1 \le n \le 10.000$) determină descompunerea în factori primi a lui $n!$, ca în exemplul:

numar.in	factorial.out
45	2^41 3^21 5^10 7^6 11^4

numar.in	factorial.out
	13^3 17^2 19^2 23^1 29^1 31^1 37^1 41^1 43^1

Pentru stocarea descompunerii în factori primi, vom folosi containerul asociativ *std::map*, care conține perechi (p, m), în care p sunt numere prime, iar m sunt puterile acestora în descompunerea în factori primi a lui $n!$. Pentru fiecare factor prim i al tuturor numerelor de la 2 la n, dacă acesta nu se află deja în map (*m.find(i)==m.end()*), atunci este adăugată acesteia perechea $(i, 1)$; este prima apariție a acestui factor în produs. Altfel, se incrementează valoarea cu cheia i:

```
mIt = m.find(i);
++(mIt->second);
```

Program 8

```
#include <map>
#include <fstream>

int main(){

  typedef std::pair<int, int> TIntPair;

  std::ifstream in("numar.in");
  std::ofstream out("factorial.out");
  std::map<int, int> m;
  std::map<int, int>::iterator mIt;
  int n;

  if( in && !in.eof() && in>>n){
    for(int k=2; k<=n; k++){
      int aux = k;
      for(int i=2; aux>1 && i<=k; ++i){
        while(aux%i == 0){
          if(m.find(i)==m.end())
            m.insert(TIntPair(i, 1));
          else {
            mIt = m.find(i);
            ++(mIt->second);
          };
```

```
            aux /= i;
          }
        }
      }
    }

  for(mIt=m.begin(); mIt != m.end(); mIt++){
    out <<  mIt->first << "^";
    out << mIt->second << std::endl;
  }

  return 0;
}
```

Problema 9. Scrieţi un program care pentru un număr natural dat n ($1 \le n \le 10$) generează toate permutările mulţimii $\{1, 2, \ldots, n\}$ în ordine lexicografică. Exemplu:

n.in	perm.out
3	1 2 3 1 3 2 2 1 3 2 3 1 3 1 2 3 2 1

Vectorul se completează mai întâi cu valorile 1, 2, ..., n. Folosim metoda *next_permutation*() din *header*-ul *<algorithm>*, care returnează *true*, dacă secvenţa specificată are o permutare succesoare, caz în care este şi transformată corespunzător.

Program 9

```
#include <vector>
#include <fstream>
#include <algorithm>

int main(){
  short n, i;
  std::vector<short> v;
  std::ifstream in("n.in");
  std::ofstream out("perm.out");
  if(in && !in.eof() && in>>n){
    for(i=1; i<=n; i++) v.push_back(i);
    do{
      for(i=0; i<v.size(); i++){
        out.width(3);
        out << v[i];
      }
```

```
    out << std::endl;
  }while(next_permutation(v.begin(), v.end()));

}
return 0;
```

Ca exercițiu, modificați programul astfel încât permutările să fie afișate în ordine lexicografică, începând cu cea mai mare dintre ele și folosind metoda *prev_permutation*().

Problema 10. Ciurul lui Eratostene este o metodă foarte cunoscută pentru determinarea numerelor prime și a fost deja prezentată în primul capitol. Așa cum specifică și numele, dintre numerele naturale se vor „cerne" acelea care nu sunt prime. Prin sită vor fi eliminați mai întâi multiplii lui 2, apoi multiplii lui 3, 5, 7 ș.a.m.d. Numerele prime care nu sunt multipli ai altor numere vor rămâne la sfârșit în sită. Scrieți un program care verifică primalitatea pentru mai multe numere naturale din intervalul [1, 5.000.000]. Exemplu:

numere.in	prim.out
123 345 12 101 43 284563 17 1006123 5000000 1021	123 -- compus 345 -- compus 12 -- compus 101 -- prim 43 -- prim 284563 -- compus 17 -- prim 1006123 -- prim 5000000 -- compus 1021 -- prim

Pentru rezolvarea problemei, vom folosi containerul *std::bitset<N>*, care stochează N biți și oferă metode pentru manipularea lor. Mai întâi, toți biții de la 0 la *MAX_SIZE* sunt setați pe 1 folosind metoda *set*() (metoda *reset*() setează toți biții pe 0). Apoi, se vor parcurge toate numerele naturale de la 2 la $\sqrt{MAX_SIZE}$ inclusiv și, dacă numărul corespunzător este prim (*sieve[i]* == 1), se vor „elimina" din sită toți multiplii lui prin setarea *sieve[i * j] = false*.

Program 10

```
#include <fstream>
#include <bitset>

const unsigned MAXSIZE = 5000000;
```

```
using namespace std;
int main(){
  unsigned long i, j, n;
  bitset<MAXSIZE+1> sieve;
  ifstream fin("numere.in");
  ofstream fout("prime.out");
  sieve.set(); sieve.at(1)=false;
  for(i=2; i*i<=MAXSIZE; i++)
    if(sieve[i]){
      j=2;
      while(i*j<=MAXSIZE){
        sieve[i*j]=false;
        j++;
      }
    }
  while(fin && !fin.eof() && fin>>n){
    if(sieve[n]) fout << n << "  -- prim\n";
    else fout << n << "  -- compus\n";
  }
  return 0;
}
```

Notă: o rezolvare în C se găsește de exemplu în [Log06], p. 142.

Întrebări, exerciții, probleme propuse

1. Ce este *STL*? Care este scopul creării sale și ce conține?
2. Ce este programarea generică? Dați exemple.
3. Ce este un container în *STL*? Enumerați metodele generale asupra tuturor containerelor.
4. Enumerați containerele secvențiale, asociative, adaptor și specializate, precum și trăsăturile fiecăruia dintre ele.
5. Ce sunt iteratorii și care este scopul lor? Cum sunt ei legați de containere și algoritmi?
6. Ce este un functor? Definiți conceptele: *generator*, *predicat* și *predicat binar*. Dați exemple de utilizare a predicatului binar.
7. *Header*-ul *<algorithm>* din *STL* conține 60 de algoritmi. Cum se pot aceștia clasifica? Dați exemple de metode din fiecare categorie.
8. Modificați programul 4 astfel încât afișarea să se facă folosind iteratori (variantele *iterator* și *reverse_iterator*) și folosind metoda *at*() (acces verificat) în locul metodei *copy*.
9. Scrieți variante C pentru problemele 5 și 6 (parcurgerile *BFS* și *DFS*).

10. Dezvoltați programul 8 astfel încât să rezolve problema determinării combinărilor de n luate câte k ($0 \leq k \leq n \leq 1000$), folosind descompuneri ca *std::map* și redând soluția în același format ca și factorialul, prin descompunerea în factori primi în ordinea crescătoare a acestora.
11. Modificați programul 9 astfel încât permutările să fie afișate în ordine colexicografică,. începând cu cea mai mare dintre ele și folosind metoda *prev_permutation*() din *<algorithm>*.
12. Un *std::bitset<N>* se diferențiază de *std::vector<bool>* prin faptul că are o lungime fixă. Modificați programul 10 astfel încât să folosiți *std::vector<bool>* în loc de *std::bitset*.
13. Modificați toate programele care folosesc tipul *std::vector*, prin înlocuirea acestuia cu tipul *std::list*.
14. Manipularea șirurilor de caractere în C++ este mai ușoară decât în C, deoarece lucrul cu memoria este rezolvat implicit, dar *STL* oferă și multe metode utile specifice. Citiți din *Help* metodele și utilizarea tipului șir de caractere *std::string*. Scrieți eventual un referat care să conțină teorie și exemple privind metodele uzuale.
15. Scrieți un program care stochează în variabile de tip *std::bitset* numere întregi date de la tastatură (de exemplu, 12, 567, – 34, 5, – 3421). Scrieți pentru aceasta două variante (una dintre ele folosind operatorii pe biți).
16. Scrieți un program demonstrativ pentru testarea comportării algoritmilor *for_each, search, transform, replace, reverse* și *swap*.

Vedere din turnul Catedralei din Ulm

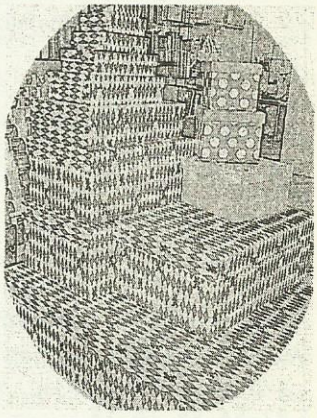
Capitolul 4
Cutii ascunse

- Descrierea problemei
- Analiza problemei şi proiectarea soluției
- Algoritmul
- Programul
- Analiza programului
- Trei mici trucuri de programare
- Probleme propuse
- Observații

Pericolul trecutului era acela că oamenii deveneau sclavi.
Pericolul viitorului este acela că ei pot deveni roboți.
Anonim

În acest capitol, prezentăm o problemă de programare dinamică. După descrierea problemei, urmează analiza ei și proiectarea unei soluții, care este descrisă și în pseudocod. Apoi implementăm un program C++, care va fi analizat din perspectiva programării orientate obiect.

Descrierea problemei



Considerăm o cutie n-dimensională. Atunci când ne referim la două dimensiuni, perechea (2, 3) poate să reprezinte, de exemplu, o cutie cu lățimea 2 și lungimea 3. Într-un spațiu tridimensional, tripletul (4, 8, 9) poate să reprezinte o cutie cu lățimea 4, lungimea 8 și înălțimea 9. Nu este ușor să ne imaginăm o cutie în mai multe dimensiuni, dar vom putea abstractiza și opera cu acest concept. Spunem că o cutie $A = (a_1, a_2, ..., a_n)$ încape în cutia $B = (b_1, b_2, ..., b_n)$, dacă există o permutare π a mulțimii $\{1, 2, ..., n\}$, astfel încât $a_{\pi(i)} < b_i$ pentru toți $i \in \{1, 2, ..., n\}$. Aceasta înseamnă că ordinea dimensiunilor unei cutii poate fi schimbată aleatoriu (cutia se poate contorsiona). Dorim găsirea celui mai lung subșir de cutii, care încap una în alta. Cutiile $C_1, C_2, ..., C_k$ reprezintă un astfel de subșir atunci când fiecare cutie C_i încape în cutia C_{i+1} (pentru orice i astfel încât $1 \leq i < k$). De exemplu, cutia $A = (2, 6)$ încape în cutia $B = (7, 3)$, deoarece dimensiunile cutiei A pot fi permutate $A = (6, 2)$ și fiecare dimensiune este mai mică decât cea corespunzătoare în B. Cutia $A = (9, 5, 7, 3)$ nu încape în cutia $B = (2, 10, 6, 8)$, deoarece nu există nici o permutare a dimensiunilor cutiei A care să îndeplinească această condiție. Dar cutia $C = (9, 5, 7, 1)$ încape în cutia B, deoarece îi putem permuta dimensiunile la $(1, 9, 5, 7)$ și fiecare dintre ele este mai mică decât dimensiunea corespunzătoare din B.

Date de intrare. În fișierul *cutii.in* se găsesc mai multe șiruri de cutii. Fiecare astfel de șir începe cu o linie pe care se află numărul k de cutii și dimensiunea n a acestora. Următoarele k linii conțin fiecare câte n dimensiuni corespunzătoare unei cutii. Dimensiunea maximă a unei cutii este 250, cea minimă 1. Numărul maxim de cutii într-o secvență dată este 300. Considerăm că datele de intrare sunt corecte!

Date de ieșire. Trebuie deci să găsim un subșir cu număr maxim de cutii care încap una în alta. Pentru cazul când există mai multe subșiruri cu același număr maximal de cutii, se cere doar unul, în forma din *cutii.out*. Exemplu:

cutii.in	cutii.out
5 2 3 7 8 10 5 2 12 7 21 18 **8 6** 5 2 20 1 30 10 23 15 7 9 11 3 40 50 34 24 14 4 9 10 11 12 13 14 31 4 18 8 27 17 44 32 13 19 41 19 1 2 3 4 5 6 80 37 47 18 21 9 **9 5** 7 14 2 1 3 49 80 15 50 10 90 53 17 60 11 4 3 2 15 10 1 2 3 4 5 6 7 8 9 10 89 53 17 60 11 3 2 1 14 9 92 54 65 19 15	Lungime: 4 -------------- 3 1 4 5 ********** Lungime: 4 -------------- 7 2 5 8 ********** Lungime: 5 -------------- 5 4 2 7 9 **********

(ACM Internet Programming Contest 1990, Problem D. Stacking Boxes)

Analiza problemei și proiectarea soluției

Propoziția 1. Se consideră două cutii n-dimensionale $A = (a_1, a_2, \ldots, a_{n-1}, a_n)$ și $B = (b_1, b_2, \ldots, b_{n-1}, b_n)$, cu proprietatea că $a_i \leq a_{i+1}$, $b_i \leq b_{i+1}$ pentru orice i de la 1 la $n-1$ (dimensiunile sunt sortate crescător). *Atunci, cutia A încape în cutia B* dacă și numai dacă $a_i < b_i$ pentru orice $i \in \{1, 2, \ldots, n\}$.

Demonstrație. Folosim metoda reducerii la absurd. Presupunem că A încape în B și că există un $k \in \{1, 2, ..., n\}$, astfel încât $a_k \geq b_k$. Considerăm cel mai mic număr natural k cu această proprietate: $a_i < b_i$ pentru toți $i \in \{1, 2, ..., k-1\}$ și $a_k \geq b_k$. Deoarece (a_i) este un șir crescător, rezultă că $a_i \geq b_k$ pentru toți $i \in \{k+1, ..., n\}$. Singura posibilitate ca la poziția k să aibă loc $a_k < b_k$ este interschimbarea elementului a_k cu una dintre valorile $\{a_1, a_2, ..., a_{k-1}\}$. Considerăm că aceasta este la poziția j. Pentru j va avea loc atunci $a_j \geq b_j$, de unde rezultă că A nu încape în B. Contradicție! În cealaltă direcție, propoziția este adevărată prin definiție. ❑

Primul pas în proiectarea algoritmului este deci sortarea crescătoare a dimensiunilor fiecărei cutii. Al doilea pas va fi ordonarea lexicografică a cutiilor, cu memorarea pozițiilor inițiale. Pentru ca o cutie A să încapă într-o altă cutie B, este necesar (nu și suficient!), ca A să se afle înaintea lui B în acest șir ordonat lexicografic (o cutie de pe o poziție mai mică poate să încapă într-o cutie de pe o poziție mai mare, dar invers este imposibil!). După aceste prelucrări, problema se reduce la determinarea subșirului maximal crescător. Condiția de comparație ≤ devine acum *încape*. Pentru primul set de date de intrare din *cutii.in*, se vor executa următorii pași:

1. Ordonarea crescătoare a dimensiunilor pentru fiecare cutie		2. Sortarea lexicografică a cutiilor cu memorarea poziției inițiale	
3 7 8 10 5 2 12 7 21 18	→	3 7 8 10 2 5 7 12 18 21	→
		3 7 8 10 2 5 7 12 18 21	2 5 (3) 3 7 (1) 7 12 (4) 8 10 (2) 18 21 (5)

Determinarea subșirului maximal crescător este o problemă clasică de programare dinamică și este tratată și în capitolul 11. Pentru exemplul nostru, subșirul crescător maximal cu condiția *încape* este (2, 5) → (3, 7) → (7, 12) → (18, 21), singurul cu lungimea 4. Pozițiile inițiale ale cutiilor corespunzătoare (3 1 4 5) se vor scrie în fișierul de ieșire.

Algoritmul

Considerăm n obiecte de tip *Cutie* $C_1, ..., C_n$, care își cunosc dimensiunile și poziția inițială. Pentru fiecare i considerăm un subșir de cutii ce are ca ultim element cutia C_i. Pentru aceasta, vom construi doi vectori $v[]$ și $vPred[]$. Lungimea subșirului care sfârșește cu C_i este salvată în $v[i]$ și $vPred[i]$ este indexul penultimei cutii în acest subșir ($vPred[i] = -1$ și $v[i] = 1$). Deci:

- $v[i]$ este lungimea maximă a unui subșir de cutii care încap una în alta, a cărui

ultimă cutie este C_i. Formal, scriem:

– $v[1] \leftarrow 1$ (prima cutie nu poate să conţină alta, acest subşir este format numai din acest element),

– $v[i] \leftarrow 1 + \max\{v[j] \mid j < i$ şi C_j încape în $C_i\}$ (C_j este penultima cutie în acest subşir).

- *vPred*[*i*] conţine indexul *j* cu condiţia: C_j să fie cutia predecesoare în şirul maximal în care ultima cutie este C_i (dacă o cutie nu are nici o predecesoare, atunci această valoare este – 1):

  – $vPred[1] \leftarrow -1$ (prima cutie nu are nici o predecesoare),

  – $vPred[i] \leftarrow j$, $v[j]$ maximal cu ($j < i$ şi C_j încape în C_i).

Dacă există mai multe subşiruri cu aceeaşi lungime maximală, atunci se va considera doar primul dintre ele.

Vectorii *v*[] şi *vPred*[] vor fi populaţi secvenţial. Folosim variabila *imax* pentru a salva indexul actual optimal. Dacă lungimea $v[i]$ corespunzătoare poziţiei actuale este mai mare decât valoarea $v[imax]$, atunci *imax* va fi actualizat.

În acest moment putem formula algoritmul în pseudocod:

```
ALGORITM_CUTII_ASCUNSE
   1. Read Boxes C1, C2, ..., Cn
   2. For( i ← 1, n; step 1) Execute
                 Sort_Dimensions(Ci)
      End_For
   3. Sort_Lexicographical(C1, C2, ..., Cn)
   4. v[1] ← 1, vPred[1] ← – 1, imax ← 1
   5. For (i ← 2, n; step 1) Execute
      5.1. v[i] ← 1, vPred[i] ← – 1
      5.2. For (j ← 1, i – 1; step 1) Execute
                 If (Cj încape în Ci AND v[j]+1 > v[i]) Then
                       v[i] ← v[j]+1
                       vPred[i] ← j
                 End_If
           End_For
      5.3. If (v[i] > v[imax]) Then
                 imax ← i
           End_If
      End_For
   6. recoverBoxesSubstring (C[1..n], vPred[], imax)
END_ ALGORITM_CUTII_ASCUNSE
```

Metoda *recoverBoxesSubstring()* reconstituie recursiv subşirul maximal de cutii pe baza vectorului *vPred*[] şi a indexului *imax*:

```
recoverBoxesSubstring (C[1..n], vPred[], i)
        1. If (vPred[i] ≠ – 1) Then
                recoverBoxesSubstring(C[1..n], vPred[], vPred[i])
           End_If
        2. Write Original_Position(Ci)
End_ recoverBoxesSubstring ()
```

Complexitatea algoritmului este $O(n^2m + nm\log m)$, unde n reprezintă numărul cutiilor şi m dimensiunea unei cutii (din pasul 2 rezultă $O(nm\log m)$, deoarece sunt sortate n cutii; din pasul 5 rezultă $O(n^2)$, deoarece avem două bucle *for* imbricate, fiecare de lungime n; în a doua buclă *for* avem nevoie de $O(m)$ pentru a testa condiţia *incape*).

Programul

Pentru a implementa modelul matematic descris, scriem clasa *CBox*, care conţine ca membri *_n* (dimensiunea cutiei), *_i* (poziţia iniţială) şi vectorul *_vDim* (ce reţine dimensiunile). Metodele *getI()*, *getN()*, *elementAt()*, *sort()* şi *isFit()* prelucrează elemente de tipul abstract *CBox*.

```
#include <fstream>
#include <vector>
#include <algorithm>

using namespace std;

class CBox{
  short _n, _i;
  vector<short> _vDim;
public:
  CBox(vector<short> vDim, short n, short i):
       _vDim(vDim), _n(n), _i(i){};
  short getI() const {return _i;};
  short getN() const {return _n;};
  short elementAt(short i) const{
    return _vDim.at(i);
  };
```

```
  void sort();
  bool isFit(CBox *another) const;
  friend bool operator>(CBox &b1, CBox &b2);
};

void CBox::sort(){
  std::sort( _vDim.begin(), _vDim.end() );
}

bool CBox::isFit(CBox *another) const{
  bool b = true;
  if(another->getN()!=_n ) return false;
  for(short i=0; b && i<_n; i++)
    if(_vDim.at(i)>=another->elementAt(i))
      b = false;
  return b;
}

bool operator>(CBox &b1, CBox &b2){
  return
    lexicographical_compare(b2._vDim.begin(), b2._vDim.end(),
    b1._vDim.begin(), b1._vDim.end());
}
```

Testează dacă încape în altă cutie (presupunem că dimensiunile cutiilor sunt sortate crescător și folosim *Propoziția 1*):
– mărimi diferite → nu încape;
– când o dimensiune a cutiei este mai mare sau egală cu dimensiunea corespunzătoare în cealaltă cutie, atunci nu încape;
– altfel, încape.

```
istream& operator>>(istream &is, vector<CBox*> &v) {
  short n, k, j, i, aux;
  vector<short> vDim;
  if( !is ) return is;
  is >> k >> n;
  v.clear();
  for(i=0; i<k; i++){
    vDim.clear();
    for(j=0; j<n; j++){
      is >> aux; vDim.push_back(aux);
    }
    CBox *box = new CBox(vDim, n, i+1);
    box->sort();
    v.push_back(box);
  }
  return is;
}

bool isBoxSmaller(CBox* b1, CBox *b2){
  return !(*b1 > *b2);
}
```

Citirea șirului de cutii (supraîncărcarea operatorului „>>"):
– citește *k* (numărul de cutii) și *n* (dimensiunea unei cutii);
– citește cele *n* dimensiuni ale fiecărei cutii și le stochează în vectorul *vDim*;
– creează obiectul *CBox* cu ajutorul constructorului *CBox (vDim, n, i+1)*, sortează dimensiunile cutie cu ajutorul *sort()* și adaugă acest obiect vectorului *v*[], care va fi returnat.

```
void doProcess(vector<CBox*> &vCBoxes, vector<int>&v,
               vector<int>&vPred, int &imax){
  std::sort(vCBoxes.begin(), vCBoxes.end(), isBoxSmaller);
  v.clear(); vPred.clear();
  v.push_back(1);
  vPred.push_back(-1);
  imax = 0;
  int n=(int)vCBoxes.size();
  CBox *b1, *b2;
  for(int i=1; i<n; i++){
    v.push_back(1);
    vPred.push_back(-1);
    b1=(CBox*)vCBoxes[i];
    for(int j=0; j<i; j++){
      b2=(CBox*)vCBoxes[j];
      if(b2->isFit(b1) &&
           v[j]+1 > v[i]){
        v[i] = v[j] + 1;
        vPred[i] = j;
      }
    }
    if(v[i] > v[imax]) imax = i;
  }

void recoverBoxesSubstring(vector<CBox*> &vCBoxes,
                           vector<int>&vPred,
                int i, ofstream& out){
  if(vPred[i]+1)
    recoverBoxesSubstring(vCBoxes, vPred, vPred[i], out);
  CBox* b = (CBox*)vCBoxes[i];
  out << b->getI() << " ";
}

int main(){
  vector<CBox*> vCBoxes;
  vector<int> v, vPred;
  int imax;
  ifstream in("cutii.in");
  ofstream out("cutii.out");
  while(in && !in.eof() && in >> vCBoxes){
    doProcess(vCBoxes, v, vPred, imax);
    out << " Lungime: " << v[imax] << endl;
    out << "---------------" << endl;
    recoverBoxesSubstring (vCBoxes, vPred, imax, out);
    out << endl;
```

1. Sortarea lexicografică a cutiilor.
2. Pentru fiecare cutie, creăm $v[]$ şi *vPred*[] cu semnificaţiile:
   – $v[0] \leftarrow 1$;
   – $v[i] \leftarrow 1 + \max\{v[j] \mid j < i$ şi cutia j *încape* în cutia $i\}$ ($v[i]$ este lungimea maximală a unui subşir cu i ultima cutie);
   – *vPred*[0] $\leftarrow -1$ (prima cutie nu are nici un predecesor).
3. Dacă există o cutie j cu proprietatea că ea încape în cutia i, atunci: *vPred*[i] este indexul j al cutiei cu $v[j]$ maximal; dacă o astfel de cutie nu există, atunci *vPred*[i] $\leftarrow -1$.

```
    out << "*******************************" << endl;
  }
  return 0;
}
```

Analiza programului

1. Tipul abstract *CBox* este reprezentarea în C++ a conceptului de *cutie* prezentat în descrierea problemei. Clasa încapsulează date şi metode ce prelucrează într-un mod natural aceste date.
2. Metoda *doProcess()* sortează lexicografic întregul vector cu cutii. Pentru a realiza aceasta, folosim metoda *sort()* din *Standard Template Library* (*STL*), antetul *<algorithm>*. Pe lângă pointerii care mărginesc secvenţa de sortat (*vCBoxes.begin()* şi *vCBoxes.end()*), remarcăm folosirea unui al treilea parametru, obiectul *isBoxSmaller()*, de tip funcţie. Acesta defineşte criteriul de comparaţie, care trebuie să fie satisfăcut de două elemente consecutive din secvenţa sortată. Un astfel de predicat binar este definit de către utilizator, are două argumente şi returnează *true* atunci când criteriul este adevărat şi *false*, în caz contrar.

```
std::sort(vCBoxes.begin(), vCBoxes.end(), isBoxSmaller);
```

3. Metoda *std::sort()* va fi folosită din nou, de această dată cu doi parametri:

```
void CBox::sort(){
  std::sort(_vDim.begin(), _vDim.end());
}
```

   Se presupune că criteriul de comparaţie este operatorul implicit „≤".

4. În metoda *isBoxSmaller()*, observăm folosirea „>" pentru a compara două obiecte de tip *CBox*. Aceasta funcţionează, deoarece acest operator este supraîncărcat (ca metodă prietenă – *friend* – locală a *operator>()*). Această metodă trebuie să fie declarată ca metodă prietenă – *friend* – a clasei *CBox*, deoarece ea are nevoie să acceseze membrul privat *_vDim*.
5. În metoda *operator>()*, observăm folosirea metodei *std::lexicographical_compare()*, care se află tot în *header*-ul *<algorithm>* al *STL*. Această metodă compară fiecare element al primei secvenţe cu elementul corespunzător al celei de-a doua secvenţe, pentru a decide care este mai mică lexicografic.

Trei mici trucuri de programare

1. Apelul repetat al aceleiaşi funcţii în condiţia unei bucle

Exemplu. Dacă dorim prelucrarea elementelor unui vector:

```
vector<int>::iterator itBegin = V.begin();
vector<int>::iterator it;
for(it = V.end()-1; it != itBegin; --it)…
```

În acest caz, este mai bine să folosim, ca mai sus, o nouă variabilă *itBegin* şi să o iniţializăm cu *V.begin()*, decât să folosim mereu apelul funcţiei *V.begin()*:

```
vector<int>::iterator it;
for(it = V.end()-1; it != V.begin(); --it)
```

Motivul este că, în prima variantă, apelul funcţiei *begin()* este executat doar o dată, iar pentru containere de dimensiune mai mare salvează timp.

2. Condiţie într-o expresie booleană

De multe ori, folosim în carte expresii de forma `if(0==n)...`, `while(1==i && 5==j)` etc., în locul expresiilor „naturale" `if(n==0)...`, `while(i==1 && j==5)` etc. Ultimele sunt echivalente cu primele, dar este recomandată utilizarea primei forme. Deseori se uită un semn de egalitate şi, în cazul primei forme, compilatorul va indica acest lucru, pe când, în al doilea caz, va compila, dar efectul nu va fi acelaşi (atribuire în loc de compararea egalităţii).

3. Instrucţiune `break` într-o instrucţiune de selecţie multiplă `switch`

Ambele utilizări de mai jos sunt echivalente:

```
switch((p2-p5)%4){
  case 0: uc = (uc*6)%10; break;
  case 1: uc = (uc*2)%10; break;
  case 2: uc = (uc*4)%10; break;
  case 3: uc = (uc*8)%10; break;
}
```

```
switch((p2-p5)%4){
  case 0: uc = (uc*6)%10; break;
  case 1: uc = (uc*2)%10; break;
  case 2: uc = (uc*4)%10; break;
  case 3: uc = (uc*8)%10;
}
```

De obicei, este mai bine să se încheie ultimul caz de decizie tot cu o instrucţiune *break*. Dacă ulterior se dezvoltă programul şi apare un nou caz în instrucţiunea *switch*, atunci nu este posibilă neintroducerea lui din neatenţie.

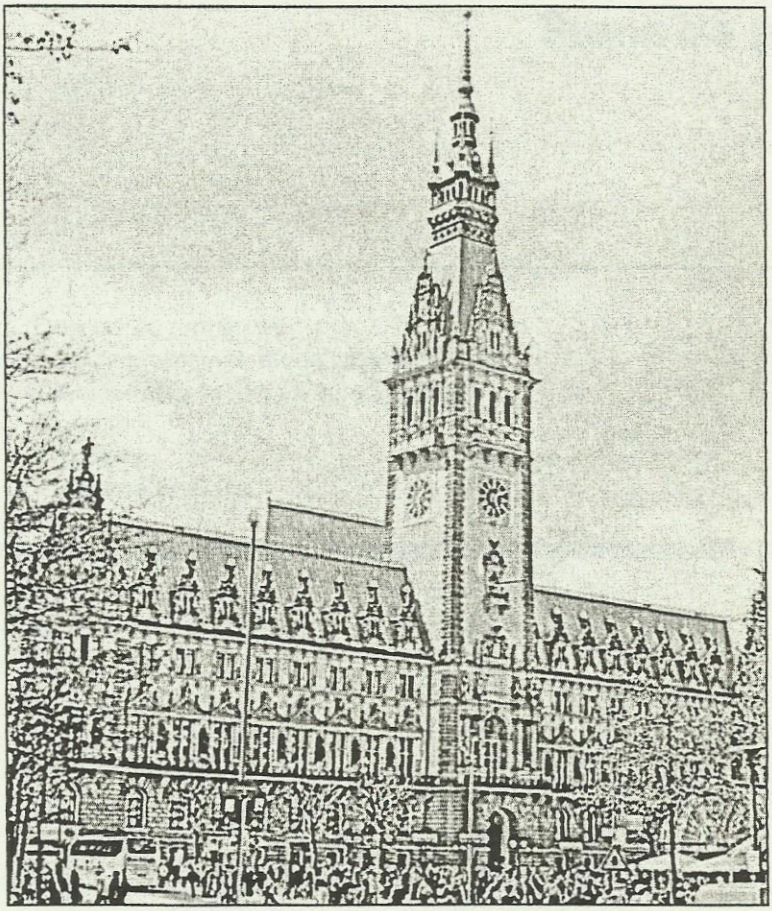
Probleme propuse

1. Într-un şir ordonat lexicografic de cutii (ale căror dimensiuni sunt sortate), o condiție necesară pentru ca o cutie *A* să încapă într-o cutie *B* este ca *A* să se afle înaintea lui *B* în şir. De ce nu este această condiție şi suficientă? Găsiți un contraexemplu.
2. Eliminați declarația de funcție prietenă (*friend*) a metodei *operator>()* din clasa *CBox*. Ce eroare furnizează compilatorul? Modificați programul astfel încât această metodă să fie declarată doar extern local.
3. Implementați iterativ metoda *recoverBoxesSubstring()*.
4. Se poate întâmpla ca datele de intrare să nu fie corecte (caractere false, lungimea unui şir de cutii prea mare sau negativă etc.). Dezvoltați programul astfel încât să se verifice şi corectitudinea datelor de intrare.
5. Scrieți un program care generează fişiere de intrare cu mai multe cazuri şi folosiți-le în program. Scrieți o metodă inteligentă, care pentru dimensiuni mai mari să aibă şi soluții subşiruri de dimensiuni mai mari.
6. Se poate ca pentru o soluție de lungime maximală să existe mai multe subşiruri de cutii care au aceeaşi lungime. Găsiți toate subşirurile de lungime maximală.
7. Scrieți un program C care rezolvă problema, pentru a compara programarea procedurală cu cea orientată obiect.
8. Găsiți motivele pentru care algoritmul *Cutii_Ascunse* este unul de programare dinamică, şi nu unul de tip *Greedy*, *Backtracking* sau *Divide-et-Impera*.

Observații

În acest capitol am folosit:

1. metoda programării dinamice;
2. recursivitatea;
3. concepte POO: abstractizarea datelor, supraîncărcarea operatorilor, funcții prietene (*friend*), funcții membru *const*, polimorfie;
4. clasa container *std::vector* din *STL* din antetul *<vector>*, cu metodele: *pop_back()*, *push_back()*, operatorul de acces *[]*, *clear()*, *begin()*, *end()*, *at()*;
5. metode din *STL*, antetul *<algorithm>*, *sort()*, *lexicographical_compare()*;
6. fluxuri de intrare şi de ieşire pentru citirea şi scrierea în/din fişiere.



Primăria din Hamburg (construită între 1886 şi 1897)

Capitolul 5
Sume de puteri

- Descrierea problemei
- Analiza problemei. Modelarea algebrică
- De la formula recursivă spre algoritm
- Algoritmul
- Programul
- Probleme propuse

Abia atunci când ne îndepărtăm de munți
îi putem vedea în adevărata lor măreție; așa e și cu prietenii...
Hans Christian Andersen

Descrierea problemei

Fie numărul natural $k \geq 0$. Suma de puteri de grad k pentru n se definește astfel:

$$S_k(n) = 1^k + 2^k + 3^k + \ldots + n^k \tag{1}$$

Se poate demonstra (cu ajutorul inducției matematice și al formulei (13) de mai jos), că $S_k(n)$ este un polinom de grad $(k+1)$ în n, cu n coeficienți raționali, deci $S_k(n)$ se poate scrie sub forma:

$$S_k(n) = \frac{1}{M}(a_{k+1}n^{k+1} + a_k n^k + \ldots + a_1 n + a_0) \tag{2}$$

M și $a_{k+1}, a_k, \ldots, a_1, a_0$ sunt numere întregi, M este număr natural. Cu aceste condiții și cu condiția ca M să fie minimal, pentru fiecare număr natural k există exact o soluție $(M, a_{k+1}, a_k, \ldots, a_1, a_0)$ a problemei. Se cere determinarea acestei soluții pentru un număr natural k dat.

Date de intrare. În fișierul *psum.in* se găsesc unul sau mai multe numere naturale k ($0 \leq k \leq 20$).

Date de ieșire. Scrieți în fișierul *psum.out* numerele $M, a_{k+1}, a_k, \ldots, a_1, a_0$ pe câte o linie pentru fiecare caz de intrare, cu M minimal, ca în exemplul:

psum.in	psum.out
2 5 0 16	6 2 3 1 0 12 2 6 5 0 -1 0 0 1 1 0 510 30 255 680 0 -2380 0 8840 0 -24310 0 44200 0 -46988 0 23800 0 -3617 0

(ACM North-Eastern European Regional Programming Contest, 1997-1998)

Analiza problemei. Modelarea algebrică

Primele sume $S_k(n)$ sunt:

$$S_0(n)=\sum_{i=1}^{n} i^0 = n \tag{3}$$

$$S_1(n)\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{1}{2}(n^2+n) \tag{4}$$

$$S_2(n) = \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} = \frac{1}{6}(2n^3+3n^2+n) \tag{5}$$

$$S_3(n) = \sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4} = \frac{1}{4}(n^4+2n^3+n^2) \tag{6}$$

$$S_4(n) = \sum_{i=1}^{n} i^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30} = \frac{1}{30}(6n^5+15n^4+10n^3-n) \tag{7}$$

Aceste formule sunt relativ cunoscute. Ele pot fi demonstrate, de exemplu, prin inducție şi ne dorim mai departe determinarea lui $S_5(n)$. Cu ajutorul binomului lui Newton obținem:

$$(i-1)^6 = i^6 - C_6^1 i^5 + C_6^2 i^4 - C_6^3 i^3 + C_6^4 i^2 - C_6^5 i + 1, \tag{8}$$

echivalent cu:

$$i^6 - (i-1)^6 = 6i^5 - 15i^4 + 20i^3 - 15i^2 + 6i - 1 \tag{9}$$

Prin sumarea de la 1 la n, se obține:

$$\sum_{i=1}^{n}(i^6-(i-1)^6) = 6\sum_{i=1}^{n} i^5 - 15\sum_{i=1}^{n} i^4 + 20\sum_{i=1}^{n} i^3 - 15\sum_{i=1}^{n} i^2 + 6\sum_{i=1}^{n} i - \sum_{i=1}^{n} 1 \tag{10}$$

Reducem termenii în membrul stâng:

$$n^6 = 6S_5(n) - 15S_4(n) + 20S_3(n) - 15S_2(n) + 6S_1(n) - S_0(n) \tag{11}$$

Din formula (11) se obține imediat:

$$S_5(n)=\frac{n^2(n+1)^2(2n^2+2n-1)}{12}=\frac{1}{12}(2n^6+6n^5+5n^4-n^2) \tag{12}$$

Generalizarea acestui exemplu concret ne conduce la:

$$(k+1)S_k(n)=n^{k+1}+C_{k+1}^2\,S_{k-1}(n)-C_{k+1}^3\,S_{k-2}(n)+\ldots+(-1)^pC_{k+1}^p\,S_{k-p+1}(n)+\ldots$$
$$+(-1)^kC_{k+1}^k\,S_1(n)+(-1)^{k+1}S_0(n) \tag{13}$$

De la formula recursivă spre algoritm

Pe baza formulei (13), se deduce că, pentru a calcula soluția $(M, a_{k+1}, a_k, \ldots, a_1, a_0)$ a problemei pentru un k dat, este nevoie de soluțiile tuturor problemelor pentru dimensiuni mai mici decât k, anume 0, 1, ..., $k-1$. Toate aceste informații vor fi secvențial calculate şi salvate, pentru a putea fi folosite ulterior. Vom folosi un tablou M[] pentru a salva valorile $M[1]$, $M[2]$,..., $M[k]$ cu condiția: $M[i]$ este numărul natural M din descrierea problemei pentru $S_i(n)$, $0 \le i \le k$.

În tabloul bidimensional A[][], salvăm valorile $(a_{i+1}, a_i, \ldots, a_1, a_0)$, pentru $0 \le i \le k$, cu semnificația că linia i conține coeficienții corespunzători pentru $S_i(n)$.

Vizualizarea tabelară a matricei A[][] şi a vectorului M[]

A[0][0]	A[0][1]							M[0]
A[1][0]	A[1][1]	A[1][2]						M[1]
A[2][0]	A[2][1]	A[2][2]	A[2][3]					M[2]
...	...			...	...			...
A[k-1][0]	A[k-1][1]	A[k1][2]	A[k-1][3]	A[k-1][4]	...	A[k-1][k]		M[k-1]
A[k][0]	**A[k][1]**	**A[k][2]**	**A[k][3]**	**A[k][4]**	**...**	**A[k][k]**	**A[k][k+1]**	**M[k]**

Cu această reprezentare şi formula (2) rezultă:

$$S_k(n)=\frac{1}{M[k]}\left(A[k][k+1]n^{k+1}+A[k][k]n^k+\ldots+A[k][1]n+A[k][0]\right) \tag{14}$$

Folosim acum relația recursivă (13) pentru a determina secvența $(M[k], A[k][k+1], A[k][k], \ldots, A[k][1], A[k][0])$ cu ajutorul secvențelor $(M[i], A[i][i+1], A[i][i], \ldots, A[i][1], A[i][0])$ pentru $0 \le i < k$. La prima vedere pare să fie foarte dificil, dar vom vedea cum prin transformări succesive formula se simplifică. Prin substituirea cu tablouri a valorilor $a_0, a_1, \ldots, a_k$ şi M din formula (13) obținem:

$$(k+1)S_k(n) = n^{k+1}$$

$$+ C_{k+1}^2\left(\frac{1}{M[k-1]}(A[k-1][k]n^k + A[k-1][k-1]n^{k-1} + \ldots + A[k-1][1]n + A[k-1][0])\right)$$

$$- C_{k+1}^3\left(\frac{1}{M[k-2]}(A[k-2][k-1]n^{k-1} + A[k-2][k-2]n^{k-2} + \ldots + A[k-2][1]n + A[k-2][0])\right)$$

$$+ \ldots + \tag{15}$$

$$(-1)^p C_{k+1}^p\left(\frac{1}{M[k+1-p]}(A[k+1-p][k+2-p]n^{k+2-p} + \ldots + A[k+1-p][1]n + A[k+1-p][0])\right)$$

$$+ \ldots$$

$$(-1)^k C_{k+1}^k\left(\frac{1}{M[1]}(A[1][2]n^2 + A[1][1]n + A[1][0])\right) +$$

$$(-1)^{k+1} C_{k+1}^{k+1}\left(\frac{1}{M[0]}(A[0][1]n + A[0][0])\right)$$

Grupăm termenii după puterile lui n în forma unui polinom de grad $(k+1)$:

$$(k+1)S_k(n) = n^{k+1} +$$

$$n^k\left(\frac{C_{k+1}^2}{M[k-1]}A[k-1][k]\right) +$$

$$n^{k-1}\left(\frac{C_{k+1}^2}{M[k\text{-}1]}A[k-1][k\text{-}1] - \frac{C_{k+1}^3}{M[k-2]}A[k-2][k-1]\right) +$$

$$n^{k-2}\left(\frac{C_{k+1}^2}{M[k-1]}A[k-1][k-2] - \frac{C_{k+1}^3}{M[k-2]}A[k-2][k-2] + \frac{C_{k+1}^4}{M[k-3]}A[k-3][k-2]\right) + \tag{16}$$

$$\ldots$$

$$n^p\left(\frac{C_{k+1}^2}{M[k-1]}A[k-1][p] - \frac{C_{k+1}^3}{M[k-2]}A[k-2][p] + \ldots + (-1)^{k+2-p}\frac{C_{k+1}^{k+2-p}}{M[p-1]}A[p-1][p]\right)\ldots$$

$$n\left(\frac{C_{k+1}^2}{M[k-1]}A[k-1][1] - \frac{C_{k+1}^3}{M[k-2]}A[k-2][1] + \ldots + (-1)^{k+1}\frac{C_{k+1}^{k+1}}{M[0]}A[0][1]\right) +$$

$$\left(\frac{C_{k+1}^2}{M[k-1]}A[k-1][0] - \frac{C_{k+1}^3}{M[k-2]}A[k-2][0] + \ldots + (-1)^{k+1}\frac{C_{k+1}^{k+1}}{M[0]}A[0][0]\right)$$

În această formulă (16), coeficienții se pot scrie ca:

$$W[t] = (-1)^{k+1-t}\frac{C_{k+1}^{k+1-t}}{M[t]}\text{ , pentru orice } t, \text{ astfel încât } 0 \le t \le k-1 \tag{17}$$

Cu ajutorul formulei (17) formula (16) se transformă în:

$$
\begin{aligned}
&(k+1)S_k(n) = n^{k+1} + \\
&n^k(W[k-1]A[k-1][k]) + \\
&n^{k-1}(W[k-1]A[k-1][k-1] + W[k-2]A[k-2][k-1]) + \\
&n^{k-2}(W[k-1]A[k-1][k-2] + W[k-2]A[k-2][k-2] + W[k-3]A[k-3][k-2]) + \\
&\ldots \\
&n^p(W[k-1]A[k-1][p] + W[k-2]A[k-2][p]) + \ldots + W[p-1]A[p-1][p]) + \\
&\ldots \\
&n(W[k-1]A[k-1][1] + W[k-2]A[k-2][1]) + \ldots + W[0]A[0][1]) + \\
&(W[k-1]A[k-1][0] + W[k-2]A[k-2][0]) + \ldots + W[0]A[0][0])
\end{aligned}
\tag{18}
$$

În acest moment, am ajuns deja la o formă mai simplă. Coeficienţii puterilor lui n în această egalitate pot fi salvaţi într-un tablou unidimensional cu fracţii raţionale $F[]$:

$$
\begin{aligned}
&F[0] = \sum_{t=k-1}^{1} W[t]A[t][0] + W[0]A[0][0] \\
&F[p] = \sum_{t=k-1}^{p-1} W[t]A[t][p] \text{ pentru orice } p \text{ cu } 1 \le p \le k \\
&F[k+1] = 1
\end{aligned}
\tag{19}
$$

Prin înlocuirea formulelor (19) în (18), se obţine:

$$S_k(n) = \frac{1}{k+1}(F[k+1]n^{k+1} + F[k]n^k + F[k-1]n^{k-1} + \ldots F[1]n + F[0]) \tag{20}$$

Am ajuns în acest fel la o formulă care este foarte asemănătoare cu cea din descrierea problemei. Trebuie doar să mai ţinem cont de determinarea soluţiei problemei cu M minimal. Aceasta se va obţine prin calcularea celui mai mic multiplu comun al tuturor numitorilor fracţiilor $F[0]$, $F[1]$, ..., $F[k]$, $F[k+1]$. Fie acesta Q. Cu ajutorul său, vom obţine din formula (20):

$$
\begin{aligned}
&S_k(n) = \frac{1}{(k+1)Q}(F'[k+1]n^{k+1} + F'[k]n^k + F'[k-1]n^{k-1} + \ldots F'[1]n + F'[0]) \\
&F'[r] = F[r] \cdot Q,\ 0 \le r \le k+1
\end{aligned}
\tag{21}
$$

Deoarece Q este cel mai mic multiplu comun al numitorilor $F[]$, rezultă că $F'[]$ conține doar numere întregi și $(k+1)Q$ este cel mai mic M pentru suma $S_k(n)$.

Algoritmul

Din reprezentarea algebrică de mai sus, rezultă că $M[0] = 1$, $A[0][1] = 1$, $A[0][0] = 0$.

Algoritmul este atunci:

```
ALGORITM_SUME_PUTERI
    1. Read k (0 ≤ k ≤ 20)
    2. genCombinations(C[k + 1][k + 1] )
    3. M[0] = 1, A[0][1] = 1, A[0][0] = 0
    4. For(p ← 1, k; step 1) Execute
        4.1. sign = 1
        4.2. For (t ← p – 1, 0; step – 1) Execute
                W[t] = simplifyFraction(sign * C[p + 1][p + 1 – t]/M[t])
                sign = sign * (– 1)
             End_For
        4.3. For (t ← p, 0; step –1) Execute
                fAux = Fraction(0/1)
                For (r ← p – 1, t – 1; r≥ 0; step – 1) Execute
                   fAux2 = simplifyFraction (W[r] * A[r][t])
                   fAux = sumFractions ( fAux, fAux2)
                End_For
                F[t] = fAux
             End_For
        4.4. fAux = simplifyFraction(W[0] * A[0][0] )
             F[0] = sumFractions(F[0], fAux);
        4.5. F[p+1] = Fraction (1/1)
        4.6. Q = lcmDenomin (F[], p + 1)
        4.7. M[p] = (p + 1) * Q
        4.8. For (t ← p + 1, 0; step – 1) Execute
                fAux2 = simplifyFraction (F[t] * Q)
                A[p][t] = Numerator(fAux)
             End_For
        End_For
    5. Write M[k], A[k][k+1], A[k][k], ..., A[k][1], A[k][0]
END_ ALGORITM_SUME_PUTERI
```

- La pasul 2 se generează toți coeficienții binomiali până la nivelul $k+1$ (conform egalității (13) pentru calculul lui $S_k(n)$, avem nevoie de coeficienții binomiali de

k+1). Acest calcul se bazează tot pe un algoritm de programare dinamică cu ajutorul formulelor:

$$C[i][0] = C[i][i] = 1 \text{ pentru } 0 \le i \le k+1.$$
$$C[i][j] = C[i-1][j-1]+C[i-1][j] \text{ pentru } 1 \le i \le k+1, 1 \le j < i$$

- La pasul 3 se inițializează valorile corespunzătoare lui $S_0(n)$ ((2) și (14)).
- La pasul 4 începe o buclă *for* în care sunt calculate secvențial valorile $A[p][0]$, $A[p][1]$, ..., $A[p][p+1]$, $M[p]$, $1 \le p \le k$, care sunt soluțiile problemei pentru 1, 2, ..., k, conform formulelor de mai sus.
- 4.1 și 4.2: fracțiile $W[]$ sunt calculate cu ajutoul formulelor (17). Metoda *simplifyFraction()* returnează fracția simplificată. Aceasta se realizează prin aflarea celui mai mare divizor comun d al numitorului și numărătorului, urmată de împărțirea acestora la d.
- Pasul 4.3 calculează fracțiile $F[]$ cu ajutorul formulei (19) și al metodei *sumFractions()* (ce returnează suma a două fracții raționale).
- Pasul 4.4. adună la $F[0]$ valoarea $W[0] \times A[0][0]$, conform primei părți a formulei (19).
- La fel, pe baza formulei (19), se inițializează fracția $F[p+1]$ cu $\frac{1}{1}$ la pasul 4.5.
- La pasul 4.6, se va calcula Q ca fiind cel mai mic multiplu comun al numitorilor fracțiilor $F[0]$, $F[1]$, ..., $F[p]$, $F[p+1]$ cu ajutorul metodei *lcmDenomin()*.
- La pasul 4.7, se calculează $M[p]$ conform (20).
- La fel, se vor calcula valorile $A[p][p+1]$, $A[p][p]$, ..., $A[p][0]$ la pasul 4.8, cu ajutorul (20).

Complexitatea și corectitudinea algoritmului. Complexitatea algoritmului este $O(n^3)$ (pasul 4 conține trei bucle *for* imbricate), deci algoritmul este polinomial. El este corect, pentru că se bazează pe succesiuni corecte de transformări matematice.

Notăm cu $P(k)$ problema determinării sumei de puteri k. Constatăm că, în cadrul algoritmului, sunt rezolvate mai întâi pe rând problemele $P(0)$, $P(1)$, ..., $P(k-1)$. Pe baza soluțiilor acestor subprobleme, se decid valorile pentru problema $P(k)$:

$$SOL_P(k) = FUNCTION(SOL_P(0), SOL_P(1), ..., SOL_P(k-1))$$

Soluția unei probleme $P(i)$ participă la aflarea soluțiilor problemelor $P(i+1)$, $P(i+2)$, ...; de aceea, soluția unei probleme se determină doar o singură dată și se salvează, pentru a fi ulterior folosită în rezolvarea problemelor de dimensiuni mai mari. Acesta este un algoritm de programare dinamică.

$A[k][0]$	$A[k][1]$	$A[k][2]$	$A[k][3]$	$A[k][4]$	$A[k][5]$	$A[k][6]$	$M[k]$	
0	1						1	← Soluția $k = 0$
0	1	1					2	← Soluția $k = 1$
0	1	3	2				6	← Soluția $k = 2$
0	0	1	2	1			4	← Soluția $k = 3$
0	-1	0	10	15	6		30	← Soluția $k = 4$
0	**0**	**-1**	**0**	**5**	**6**	**2**	**12**	← Soluția $k = 5$

Dependența. Fiecare element a_i pentru o putere k dată depinde de valorile a_i (dacă acestea există) pentru puterile k–1, k–2, ...; M depinde de toate aceste elemente, fiind un factor de normalizare al formulei.

Programul

```
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>

const MAX_NO = 20;

using namespace std;

typedef pair<long, long> TFraction;

long cmmdc(long a,  long b){
  long r;
  while(b != 0){
    r = a % b;
    a = b;
    b = r;
  }
  return a;
}

void simplifyFraction(TFraction &f){
  long d;
  d = cmmdc(f.first, f.second);
  if(!d){
    cout<<"Eroare!!! Impartire la 0!";
    return;
  }
```

Algoritmul lui Euclid: determinarea celui mai mare divizor comun pentru două numere naturale.

```
f.first /= d; f.second /= d;
if(f.second < 0){
  f.second *= -1;
  f.first *= -1;
}
```

– determinarea celui mai mare divizor comun *d* al numitorului şi numărătorului;
– simplificarea cu *d*.

```
TFraction sumFractions(TFraction& f1, TFraction& f2){
TFraction f;
simplifyFraction(f1); simplifyFraction(f2);
f.first = f1.first*f2.second + f1.second*f2.first;
f.second = f1.second*f2.second;
simplifyFraction(f);
return f;
}

long lcmDenomin(TFraction fr[MAX_NO], short n){
long  m;
short isM;
vector<long> v;
for(int i=0; i<n; i++){
  if(fr[i].first!=0)
    v.push_back(fr[i].second);
}
sort(v.begin(), v.end());
if(v.size()<1) return 1;
m = v[v.size()-1]; isM = 0;
while(!isM){
  isM = 1;
  for(size_t i=1;i<v.size()&&isM;i++)
    if(m%v[i] != 0){isM = 0; break;}
  m++;
}
m--;
return m;
}
```

Determinarea celui mai mic multiplu comun al numitorilor fracţiilor *fr*[]:
– numitorii (nenuli) sunt salvaţi într-un vector *v*;
– se începe cu cel mai mare dintre ei şi acest număr se va incrementa până când devine cel mai mic multiplu comun. (O altă metodă ar fi descompunerea în factori primi a tuturor numitorilor, fiind apoi consideraţi toţi factorii primi la puterile maxime etc.)

```
void genCombinations(long C[MAX_NO+2][MAX_NO+2], short n){
short i, j;
C[0][0] = 1;
for(i=1; i<=n; i++){
  C[i][0] = C[i][i] = 1;
  for(j=1; j<i; j++)
    C[i][j] = C[i-1][j-1]+C[i-1][j];
}
```

Coeficienţii binomiali (formula recursivă):

$C_n^k = 1$, când $k = 0$ sau $n = k$

$C_n^k = C_{n-1}^{k-1} + C_{n-1}^k$, dacă $n > k$, $k \neq 0$

```
int main(){
  long C[MAX_NO+2][MAX_NO+2];
  long A[MAX_NO+1][MAX_NO+2];
  long M[MAX_NO+1], Q;
  short k,p, t, r, sign;
  TFraction fr[MAX_NO+2], W[MAX_NO+1], fAux, fAux2;
  genCombinations(C, MAX_NO+1);
  ifstream in("psum.in");
  ofstream out("psum.out");
  M[0] = 1; A[0][1] = 1; A[0][0] = 0;
  for(p=1; p<=MAX_NO; p++){
    sign = 1;
    for(t=p-1; t>=0; t--){
      W[t].first = sign*C[p+1][p+1-t];
      W[t].second = M[t];
      sign *= -1;
      simplifyFraction(W[t]);
    }
    for(t=p; t>=0; t--){
      fAux.first = 0; fAux.second = 1;
      for(r=p-1; r>=t-1 && r>=0; r--){
        fAux2.first  = W[r].first * A[r][t];
        fAux2.second = W[r].second;
        simplifyFraction(fAux2);
        fAux = sumFractions(fAux, fAux2);
      }
      fr[t] = fAux;
    }
    fAux2.first=W[0].first*A[0][0];
    fAux2.second =W[0].second;
    simplifyFraction(fAux2);
    fr[0] = sumFractions(fr[0], fAux2);
    fr[p+1].first = 1; fr[p+1].second = 1;
    Q = lcmDenomin( fr, p+1 );
    M[p] = (p+1)*Q;
    for(t=p+1; t>=0; t--){
      fAux2.first = fr[t].first*Q;
      fAux2.second = fr[t].second;
      simplifyFraction(fAux2);
      A[p][t] = fAux2.first;
    }
  }

  while(in && !in.eof() && in>>k){
    out << M[k] << " ";
    for(t=k+1; t>=0; t--){
      out << A[k][t] << " ";
    }
```

$$S_0(n)=\sum_{i=1}^{n} i^0 = n = \frac{1}{1}(n+0)$$

$$W[t]=(-1)^{k+1-t}\frac{C_{k+1}^{k+1-t}}{M[t]},\ 0\le t\le k-1$$

$$F[0]=\sum_{t=k-1}^{1} W[t]A[t][0]+W[0]A[0][0],$$

$$F[p]=\sum_{t=k-1}^{p-1} W[t]A[t][p],$$

pentru orice p, a.î. $1\le p\le k$

$$F[k+1]=1$$

$$S_k(n)=\frac{1}{(k+1)Q}(F'[k+1]n^{k+1}+F'[k]n^k+$$

$$+F'[k-1]n^{k-1}+\ldots F'[1]n+F'[0])$$

$$F'[r]=F[r]\cdot Q,\ 0\le r\le k+1$$

```
    out << endl;
  }
  return 0;
```

Probleme propuse

1. Demonstrați formulele (3)-(7) prin inducție.
2. Citiți în *Help* metodele, structura şi utilizarea tipului *std::pair* (folosit pentru reprezentarea unei fracții).
3. Tablourile bidimensionale *A*[][] şi *C*[][] folosesc doar partea de sub diagonala principală. Optimizați spațiul folosit prin utilizarea în loc a doi vectori.
4. Considerăm că sumele de puteri se reprezintă în forma:

$$S_k(n) = F_{k+1}n^{k+1} + F_k n^k + \ldots + F_1 n + F_0,$$

unde F_{k+1}, F_k, ..., F_0 sunt fracții simplificate; de exemplu:

$$S_0(n) = \sum_{i=1}^{n} i^0 = n$$

$$S_1(n) = \sum_{i=1}^{n} i = \frac{1}{2}n^2 + \frac{1}{2}n$$

$$S_2(n) = \sum_{i=1}^{n} i^2 = \frac{1}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}n$$

Actualizați modelarea algebrică cu această reprezentare şi scrieți programul corespunzător. Exemplu:

psum.in	psum.out
2 5 0	1/3 1/2 1/6 0 1/6 1/2 5/12 0 -1/12 0 0 1 0

München – fan bavarez la Campionatul Mondial de Fotbal (Germania, 2006)

Capitolul 6
Greedy

5 probleme complet rezolvate, 14 probleme propuse

- Fundamente
- Problema fracţionară a rucsacului
- Colorarea hărţii
- Calul pe tabla de şah
- Arborele parţial de cost minim (algoritmul lui Kruskal)
- Codificarea Huffman

Când vezi că ai aceeaşi părere cu majoritatea,
e bine să mai reflectezi o dată.
Mark Twain

Fundamente

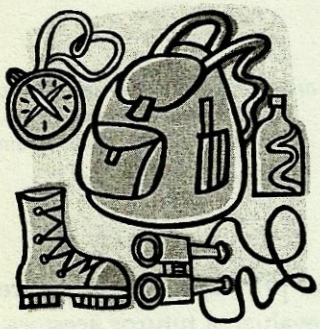
Algoritmii *Greedy* (engl. *greedy* = „lacom") se caracterizează prin faptul că, la fiecare nivel, se alege mereu cel mai bun candidat posibil, după evaluarea tuturor acestora. Aceşti algoritmi sunt, de obicei, rapizi şi furnizează o soluţie relativ bună, ceea ce nu înseamnă însă că este mereu cea mai bună soluţie. Acest lucru se observă uşor atunci când se aplică algoritmul pentru probleme clasice, cum ar fi: *problema discretă a rucsacului* sau *problema comisului voiajor*. Pentru aceste probleme se găsesc soluţii bune, dar o soluţie optimă poate fi obţinută doar prin aplicarea unei tehnici de tip *Backtracking*, caz în care timpul de execuţie creşte însă enorm. Ambele probleme sunt NP-complete, atât *Greedy*, cât şi *Backtracking* construiesc soluţia succesiv, dar numai în cazul metodei *Backtracking* se revine mereu înapoi pe un nivel predecesor, ceea ce explică timpul foarte mare de execuţie în comparaţie cu *Greedy*.

Forma generală a algoritmului *Greedy*:

```
ALGORITM_Greedy( S)
   S1  ← S
   SOL ← ∅
   While (NOT STOP-Condition) Do
     x ← a local, optimal element from S1
     S1 ← S1 – {x}
     If (SOL ∪ {x} satisfy conditions) Then
        SOL ← SOL ∪ {x}
     End_If
   End_While
END_ALGORITM_Greedy( S)
```

Metoda poate fi aplicată cu succes multor probleme, printre care determinarea celor mai scurte drumuri în grafuri (*Dijkstra*), determinarea arborelui minimal de acoperire (*Prim*, *Kruskal*), problema fracţionară a rucsacului, codificarea arborilor Huffmann.

Problema 1. Problema fracționară a rucsacului



Se consideră că dispunem de un rucsac cu capacitatea M și de n obiecte, definite fiecare prin greutate și valoare, care trebuie introduse în rucsac. Se cere o modalitate de a umple rucsacul cu obiecte, astfel încât valoarea totală să fie maximală. Este permis ca oricare obiecte și bucăți din acestea să fie introduse, iar valorile lor sunt numere reale strict pozitive.

Date de intrare. În fișierul *obiecte.in* se găsesc pe prima linie capacitatea M a rucsacului și pe următoarele linii perechi (*greutate, valoare*), pentru fiecare obiect dat, câte una pe linie. Numărul maxim de obiecte este 100, iar greutățile și valorile obiectelor pot fi considerate de tip *float*.

Date de ieșire. Scrieți în fișierul *rucsac.out* o modalitate de a umple (eventual) rucsacul care are o valoare maximă, ca în exemplul:

obiecte.in	rucsac.out
41 12.34 123.99 23.45 600.54 12.78 90.67 9.34 45.32	Obiect 2: 23.45 600.54 - complet Obiect 1: 12.34 123.99 - complet Obiect 3: 12.78 90.67 - 5.21kg

Analiza problemei și proiectarea soluției

Obiectele date se vor sorta crescător după raportul *valoare/greutate* și apoi vor fi introduse în ordine inversă în rucsac, până când acesta se umple sau toate obiectele au fost epuizate. Ultimul obiect amplasat va fi, probabil, introdus parțial. Definim clasa *Object*, care are atibutele: g (greutate), v (valoare) și *idx* (indexul corespunzător ordinii inițiale). Definim variabila statică *cont* pentru a număra obiectele create la citire și a inițializa corect câmpurile *idx* ale acestora. Implementăm *getter*-ul *getV*() (necesar atunci când umplem rucsacul) și supraîncărcăm operatorii < (se referă la raportul *valoare/greutate* și este necesar pentru apelul metodei *sort()*), >> (citirea dintr-un flux de intrare a unui obiect), << (scrierea într-un flux de ieșire a unui obiect, în forma din exemplu). Metoda *readData()* citește capacitatea M a rucsacului și obiectele din fișierul de intrare, care sunt returnate în parametrii M, respectiv v. Procesarea obiectelor citite și scrierea rezultatelor se realizează cu ajutorul meotdei *processAndWrite()*. Obiectele se vor introduce succesiv în rucsac în ordine inversă și capacitatea M se micșorează corespunzător, până când, eventual, unul trebuie să fie introdus parțial; în acest caz, M devine opusul său negativ (condiție de oprire). Complexitatea algoritmului este $O(n \log n)$, datorită metodei de sortare *sort()*.

Program 1

```
#include <fstream>
#include <vector>
#include <algorithm>

using namespace std;

class Object{
  float g, v;
  short idx;
 public:
  static short cont;
  inline float const getG(){ return g;}
  bool operator<(Object&);
  friend istream& operator>>(istream&, Object&);
  friend ostream& operator<<(ostream&, Object);
};

short Object::cont = 0;

inline bool Object::operator<(Object& other){
  return v/g < other.v/other.g;
}

istream& operator>>(istream& is, Object& ob){
  is >> ob.g >> ob.v;
  ob.idx = ++Object::cont;
  return is;
}

ostream& operator<<(ostream& os, Object ob){
  os << "Obiect " << ob.idx << ": ";
  os << ob.g << " " << ob.v;
  return os;
}

void readData(vector<Object>& v, float& M){
  ifstream in("obiecte.in");
  Object ob;
  v.clear();
  if(in && !in.eof()) in>>M;
  while(in && !in.eof() && in>>ob){
    v.push_back(ob);
  }
}
```

```
void processAndWrite(vector<Object>& v, float& M){
  std::ofstream out("rucsac.out");
  sort(v.begin(), v.end());
  size_t i=v.size()-1;
  while(i>=0 && M>0){
    if(M>=v[i].getG()){
      M -= v[i].getG();
      --i;
     } else{
        M = -M;
    }
  }
  for(size_t j=v.size()-1; j>i; --j){
    out << v[j] << " - complet" << std::endl;
  }
  if(i>=0 && M<0){
    out << v[i] << " - " << -M << " kg";
  }
}

int main(){
  vector<Object> v;
  float M;
  readData(v, M);
  processAndWrite(v, M);
  return 0;
}
```

Prin adăugarea în clasa *Object* a unui destructor care afişează un mesaj, vom observa că se creează mult mai multe obiecte decât sunt necesare, datorită faptului că prin manipularea cu ajutorul vectorului se creează copii ale acestora. Analizaţi următoarea implementare, care foloseşte *std::priority_queue* şi care nu mai suferă de acest neajuns.

Program 2

```
#include <queue>
#include <fstream>

class Object{
  float g, v;
  short idx;
public:
  Object(float greutate, float valoare,short index=0)
    :g(greutate),v(valoare),idx(index){
   }
  inline float getG()const { return g;}
```

```
  inline float getV()const { return v;}
  inline short getIdx()const { return idx;}
  inline bool operator<(const Object& other) const {
    return v/g < other.v/other.g;
  }
};

class ObjectLessComparator{    // functor pentru operator<
public:
  inline bool operator()(const Object* pLeft,
                         const Object* pRight) const
    {return (*pLeft)<(*pRight);}
};

typedef std::priority_queue<Object*,std::vector<Object*>,
                    ObjectLessComparator>
    ObjectsPriorityQueue;


std::ostream& operator<<(std::ostream& os, const Object &rObj){
  os << "Object " << rObj.getIdx()<< ": ";
  os << rObj.getG() << " " << rObj.getV();
  return os;
}

void readData(ObjectsPriorityQueue &v, float& M){
  std::ifstream in("objects.in");
  float g,val;
  short idx=0;
  if(!in.eof()) in>>M;
  while(!in.eof() && (in>>g) && (in >> val) ){
    v.push(new Object(g,val,++idx));
  }
}

void processAndWrite(ObjectsPriorityQueue &v, float M){
  std::ofstream out("rucksack.out");
  while(!v.empty()) {
    Object *pObj = v.top();
    if(M>0){
      float g = pObj->getG();
      out << (*pObj);
      M -= g;
      if(M>=0){
        out << " - vollstaendig" << std::endl;
      } else{
```

```
            out << " - " << M+g<< std::endl;
        }
    }
    v.pop();
    delete pObj;
  }
}

int main(){
  ObjectsPriorityQueue v;
  float M;
  readData(v, M);
  processAndWrite(v, M);
  return 0;
}
```

Probleme propuse

1. Implementați și o variantă C a programului.
2. În cazul variantei discrete a problemei, obiectele pot fi introduse numai complet în rucsac. În acest caz, algoritmul *Greedy* nu mai furnizează soluția optimă. Găsiți un contraexemplu.

Problema 2. Colorarea hărții

Se consideră o hartă cu n ($2 \leq n \leq 20$) țări, introdusă sub forma unei matrice $a[][]$, în care $a[i][j] = 1$, dacă țările i și j sunt vecine și $a[i][j] = 0$, dacă țările i și j nu sunt vecine.

Găsiți o modalitate de a colora harta cu cât mai puține culori, astfel încât oricare două țări vecine să fie colorate diferit. Scrieți în fișierul de ieșire *culori.out* culorile determinate pentru fiecare țară, crescător de la 1 la n, ca în exemplul:

harta.in	culori.out
7 0 1 1 1 0 0 1 1 0 1 1 0 0 0 1 1 0 1 1 0 1 1 1 1 0 1 0 1 0 0 1 1 0 1 1 0 0 0 0 1 0 1 1 0 1 1 1 1 0	1 2 3 4 1 2 5

Analiza problemei și proiectarea soluției

Problema se rezolvă optimal cu ajutorul metodei *Backtracking*, dar timpul de execuție este, în acest caz, exponențial. O metodă de tip *Greedy* oferă o soluție acceptabilă, dar

găseşte mereu numărul minim de culori. Vom colora prima țară cu 0 şi apoi, succesiv, toate celelalte țări, folosind pentru fiecare dintre ele cea mai mică culoare posibilă (diferită de a țărilor vecine deja „colorate").

Program

```
#include <fstream>

int main(){
  std::ifstream fin("harta.in");
  std::ofstream fout("culori.out");
  int n, i, j, k;
  int a[50][50], col[50];
  int ok;
  fin>>n;
  for(i=0; i<n; i++)
    for(j=0; j<n; j++)
      fin>>a[i][j];
  col[0]=0;
  for(i=1; i<n; i++){
    j=-1;
    do{
      j++; ok=1;
      for(k=0; ok&&k<i; k++)
        if(1==a[k][i] && col[k]==j)
          ok=0;
      }while(!ok);
      col[i]=j;
  }
  for(i=0; i<n; i++)
    fout<<col[i]+1<<"  ";
  return 0;
}
```

– col[0] ← 0
– caută succesiv culoarea cea mai mică posibilă pentru țara *i* (dacă există o țară vecină „mai mică" cu indexul *k* şi aceeaşi culoare, deci 1==a[k][i] && col[k]==j, atunci ok ← 0).

Problemă propusă

Desenați convenabil graful corespunzător hărții date şi găsiți o colorare a sa cu patru culori (vârfurile grafului sunt țările şi între două vârfuri se trasează o muchie numai dacă țările respective sunt vecine).

Problema 3. Calul pe tabla de şah

Scrieți un program care găseşte un drum al calului pe toată tabla de şah dată de dimensiunile $m \times n$ ($4 \leq m, n \leq 100$), astfel încât fiecare câmp este păşit doar o dată.

Date de intrare. În fişierul *cal.in* se găsesc pe prima linie dimensiunile m, n ale tablei de şah şi în a doua linie coordonatele de start (*linie, coloană*).

Date de ieşire. Scrieți drumul găsit în fişierul *cal.out*, astfel încât tabla de şah să fie reprezentată ca o matrice, în care poziția de start este marcată cu 1 şi mutările calului sunt incrementate. Atunci când nu există soluție, se va scrie în fişierul de ieşire: *„Nu exista solutie"*. Exemplu:

cal.in	cal.out
5 5 2 2	19 12 7 2 21 6 1 20 17 8 11 18 13 22 3 14 5 24 9 16 25 10 15 4 23
5 5 2 3	Nu exista solutie.

Analiza problemei şi proiectarea soluției

Vom folosi regula lui Warnsdorff, care a fost prezentată de autorul ei în 1823. Folosirea regulii conduce de cele mai multe ori la determinarea unei soluții şi timpul de execuție este redus. Înainte de a alege mutarea următoare a calului, se vor număra, pentru fiecare câmp liber posibil, câte câmpuri posibile corespunzătoare acestuia există. Pentru evitarea „fundăturilor", îl vom alege pe acela cu număr minim de succesori posibili. Explicația acestei alegeri este că, pentru câmpul cu număr minim de succesori posibili, avem şanse mai mici ca în viitor să se poată ajunge aici dintr-o altă poziție.

Tabla de şah se va salva într-o matrice *a*[][]. Cele opt poziții succesoare posibile pentru un câmp vor fi definite împreună prin tablourile *dx*[] şi *dy*[] (vezi programul). Dacă ne aflăm într-un câmp (x, y), atunci cele opt câmpuri posibile unde se poate muta sunt $x + dx[i]$, $y + dy[i]$ cu $i = 0, ..., 7$. Metoda *nrAllowedSteps()* furnizează numărul de mutări posibile de la o poziție dată (câmpuri care se află pe tablă şi nu au fost atinse). Cel mai bun succesor pentru un câmp dat se află folosind metoda *findBestSucc()*: toate câmpurile succesoare posibile sunt verificate şi cel cu număr minim de câmpuri succesoare posibile este returnat. Metoda returnează *true* atunci când există cel puțin un câmp succesor şi *false* atunci când nu există nici un succesor. Dacă un astfel de

succesor există, atunci coordonatele sale vor fi returnate în parametrii x şi y. În programul principal, se vor determina mutările calului într-o buclă *while*, în care se apelează metoda *findBestSucc()*. Variabila *ctr* numără mutările calului.

Program

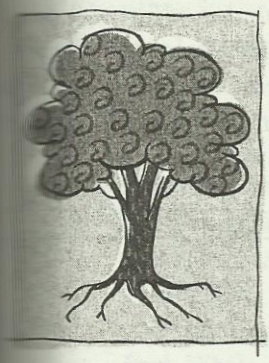
```
#include <fstream>
#include <iostream>

const short dx[8]   = {-2, -2, -1, -1, 1, 1, 2, 2};
const short dy[8]   = {-1, 1, -2, 2, -2, 2, -1, 1};
const short DIM_MAX = 100;

bool onTheTable(int x, int y, int m, int n){
  return
    (0<=x && x<m) &&
    (0<=y && y<n);
}

short nrAllowedSteps(short a[][DIM_MAX], int m, int n,
                     short x, short y)
{
  int nr=0;
  short xn, yn;
  for(int i=0; i<8; i++){
    xn=x+dx[i]; yn=y+dy[i];
    if(onTheTable(xn, yn, m, n)&&!a[xn][yn])nr++;
  }
  return nr;
}

bool findBestSucc(short a[][DIM_MAX], int m, int n,
                  short& x, short& y)
{
  int aux=INT_MAX;
  short xn, yn, xx, yy;
  for(short i=0; i<8; i++){
    xn=x+dx[i]; yn=y+dy[i];
    if(onTheTable(xn, yn, m, n)&&
      nrAllowedSteps(a,m, n, xn, yn)<aux && !a[xn][yn]){
        aux=nrAllowedSteps(a, m, n, xn, yn);
        xx=xn; yy=yn;
    }
  }
  if(aux<INT_MAX){
    x=xx; y=yy;
    return true;
```

```
  }
  return false;
}

int main(){
  short i, j, l=3, c=3, m=4, n=4;
  short a[DIM_MAX][DIM_MAX];
  std::ifstream in("cal.in");
  std::ofstream out("cal.out");
  in>>m>>n>>l>>c; l--; c--;
  for(i=0; i<m; i++)
    for(j=0; j<n; j++)
      a[i][j]=0;
  bool flag=true;
  a[l][c]=1;
  short ctr=1;
  while(flag){
     if(findBestSucc(a, m, n, l, c))
       a[l][c]=++ctr;
     else flag=false;
  }
  if(ctr==m*n){
    for(int i=0; i<m; i++){
      out << std::endl;
      for(int j=0; j<n; j++){
        out.width(4);
        out << a[i][j] <<" ";
      }
    }
  }else{
    out<<"Nu există solutie.";
  }
  return 0;
}
```

Problemă propusă

Scrieți o variantă recursivă pentru problema dată.

Problema 4. Arborele parțial de cost minim (algoritmul lui Kruskal)



Pentru a înțelege mai bine problema, vom face mai întâi o recapitulare a câtorva concepte de bază din teoria grafurilor.

Definiția 1. Un *graf* este o pereche $G = (V, E)$, în care V este o mulțime finită și nevidă de vârfuri (engl. *vertices* = „noduri"), iar E este o mulțime de perechi din $V \times V$, numite muchii (engl. *edges*).

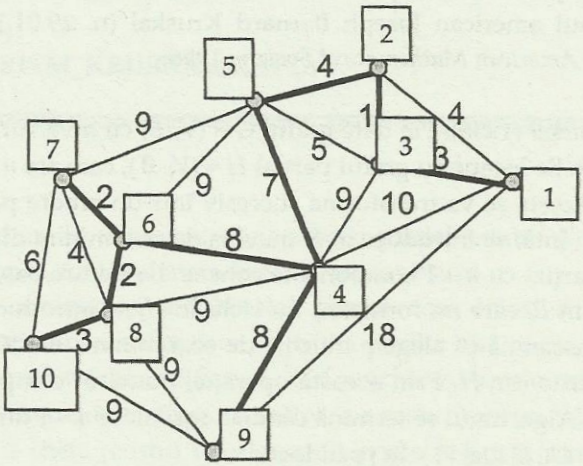
Definiția 2. O funcție $c : E \to \mathbb{R}$, prin care fiecărei muchii i se atribuie un număr real pozitiv, se numește *funcție de cost a grafului* G.

Definiția 3. Un *graf parțial* al grafului $G = (V, E)$ este un graf $H = (V, F)$, cu proprietatea că $F \subseteq E$ (H are toate vârfurile lui G, dar nu neapărat toate muchiile lui G).

Definiția 4. Fie $H = (V, F)$ un *graf parțial* al grafului $G = (V, E)$. Prin *costul* grafului parțial se înțelege suma costurilor muchiilor sale: $c(H) = \sum_{e \in F} c(e)$.

Definiția 5. Un graf parțial $H = (V, F)$ al grafului $G = (V, E)$, care conține toate vârfurile lui G și este arbore, se numește *arbore parțial al lui G*.

Definiția 6. Un arbore parțial având costul minim peste mulțimea arborilor parțiali ai unui graf dat G se numește *arbore parțial de cost minim*.



Arbore parțial de cost minim

Se dă un graf, împreună cu o funcție de cost asociată și se cere determinarea arborelui parțial de cost minim.

Date de intrare. În fișierul *kruskal.in* se găsește un graf neorientat dat prin matricea costurilor astfel: pe prima linie numărul n al vârfurilor ($1 \le n \le 200$), iar pe următoarele n linii matricea pătratică reprezentând costurile muchiilor, $c[i][j] > 0$ costul muchiei (i, j), $c[i][j] = 0$, dacă nu există o muchie (i, j). Valorile $c[i][j]$ sunt numere reale.
Date de ieșire. Scrieți rezultatele în fișierul de ieșire *kruskal.out*, câte o linie cu perechi de muchii și costul lor, urmate de costul total al arborelui parțial minim determinat, ca în exemplu:

kruskal.in	kruskal.out
10 0 4 3 10 0 0 0 0 18 0 4 0 1 0 5 0 0 0 0 0 3 1 0 9 5 0 0 0 0 0 10 0 9 0 7 8 0 9 8 0 0 4 5 7 0 9 9 0 0 0 0 0 0 8 9 0 2 2 0 0 0 0 0 0 9 2 0 4 0 6 0 0 0 9 0 2 4 0 9 3 18 0 0 8 0 0 0 9 0 9 0 0 0 0 0 0 6 3 9 0	(2, 3) -> 1 (6, 7) -> 2 (6, 8) -> 2 (1, 3) -> 3 (8, 10) -> 3 (2, 5) -> 5 (4, 5) -> 7 (4, 6) -> 8 (4, 9) -> 8 -------------------- Cost: 39

Analiza problemei și proiectarea soluției

Cei mai cunoscuți algoritmi pentru determinarea arborelui parțial de cost minim sunt cei ai lui Kruskal și Prim. Vom prezenta în continuare algorimul lui Kruskal, propus de matematicianul american Joseph Bernard Kruskal (n. 29.01.1928, New York) în *Proceedings of the American Mathematical Society*, 1956.

Algoritmul lui Kruskal (1956). Fie date graful $G = (V, E)$ cu n vârfuri și o funcție de cost asociată $c : E \to \mathbb{R}$. Se începe cu graful parțial $H = (V, \emptyset)$, care are n componente conexe (este o *pădure*). Acesta se va transforma succesiv într-un arbore parțial cu n vârfuri și $n - 1$ muchii. Mai întâi se introduce în H muchia de cost minim din G, pas prin care se obține un graf parțial cu $n - 1$ componente conexe. La fiecare pas, se va alege muchia de cost minim din E, care nu formează un ciclu în H, se introduce în H și se elimină din E. Aceasta înseamnă că alegem muchia de cost minim, ale cărei vârfuri se află în componente diferite din H. Prin această operație, numărul componentelor conexe se micșorează cu 1. Algoritmul se termină când au fost alese $n - 1$ muchii. Complexitatea algoritmului este $O(|E| \log |V|)$. În pseudocod:

```
ALGORITM_KRUSKAL(Graf G)
  SortCrescător(e1, e2, ..., em)
  For (i←1, n; step 1) Execute
    C[i] ← i                                  // inițializarea componentelor
  End_For
  cost ← 0, H ←∅
  muchiiSelectate ← 0
  While (muchiiSelectate < n – 1) Do
    (u, v) ← urmMuchie(E)
    While (C[u] ≠ C[v]) Do
      (u, v) ←urmMuchie (E)
    End_While
    H.add((u, v))                             // adaugă muchia (u, v) la H
    cost ← cost + c(u, v)
    muchiiSelectate ←muchiiSelectate + 1
    max ← max(C[u], C[v])
    min ← min(C[u], C[v])
    For (i←1, n; step 1) Execute              // unirea a două componente
      If(C[i] = max) Then C[i]← min End_If
    End_For
  End_While
  return H, cost
END_ALGORITM_KRUSKAL(Graf G)
```

Algoritmul conduce la obținerea unui arbore parțial minim şi aceasta se poate demonstra, de exemplu, folosind metoda reducerii la absurd.

Pentru salvarea grafului şi a arborelui parțial vom folosi o structură de tip *std::multimap*, ce conține perechi în care pe prima poziție este costul şi pe a doua este un număr întreg reprezentând codificarea unei muchii. Avantajul este că muchiile se vor afla în *multimap* mereu în ordinea crescătoare a costurilor lor. Folosim *multimap*, şi nu *map*, pentru că o cheie (costul unei muchii) poate să apară repetat. Dacă n este numărul vârfurilor grafului, atunci muchia (i, j) se va reprezenta prin $i * n + j$ (pentru a regăsi i şi j pe baza unei codificări k: $i \leftarrow k \text{ div } n$, $j \leftarrow k \text{ mod } n$). Deoarece matricea costurilor este simetrică față de diagonala principală, vom salva în graf doar partea superioară a acesteia:

```
in >> aux;
if(i<j && aux)
  E.insert(TPair(aux, i * n + j));   // inserează în E perechea (aux, i * n + j)
```

Urmăriți în program implementarea algoritmului lui Kruskal și utilizarea tipurilor *std::multimap* și *std::pair*, a iteratorilor și a metodelor *insert* și *erase*.

Program

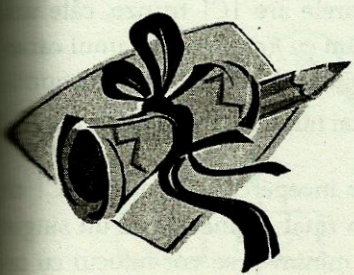
```
#include <fstream>
#include <vector>
#include <map>

using namespace std;

typedef pair<double, int> TPair;

int main(){
  ifstream in("kruskal.in");
  ofstream out("kruskal.out");
  multimap<double, int> E, H;
  double aux;
  short n;
  if(in && !in.eof() && in>>n){
    for(short i=0; in && !in.eof() && i<n; i++)
      for(short j=0; in && !in.eof() && j<n; j++){
        in >> aux;
        if(i<j && aux)
          E.insert(TPair(aux, i*n+j));
      }
  }

  H.clear();
  vector<short> C;
  for(short i=0; i<n; i++) C.push_back(i);
  while(H.size() < n-1){
    short idx=0;
    short uu, vv;
    multimap<double, int>::iterator it;
    it = E.begin();
    uu = (it->second)/n; vv = (it->second)%n; //„decodificarea" lui it->second
    while(C[uu]==C[vv]){
      it++;
      uu = it->second/n;
      vv = it->second%n;
    };
```

```
H.insert(TPair(it->first, it->second));   // inserează muchie în H
E.erase(it);                              // şterge muchie din E
short mi = min(C[uu], C[vv]);
short ma = max(C[uu], C[vv]);
for(short i=0; i<n; i++)
  if(C[i]==ma) C[i]=mi;
}

multimap<double, int>::const_iterator it;
double cost = 0;
for(it=H.begin(); it!=H.end(); it++){
  out << "(" << it->second/n+1 << ", " << it->second%n+1
      << ") -> " << it->first << endl;
  cost += it->first;
}
out << "--------------------" << endl
    << "Cost: " << cost;

return 0;
```

Probleme propuse

1. Scrieți pe hârtie paşii şi structurile intermediare pentru exemplul dat.
2. Modularizați programul prin împărțirea în metode *citireDate()*, *constrArboreKruskal()*, *afisareArbore()* şi apoi folosiți-le în programul principal.
3. Scrieți o implementare C pentru problemă.
4. Un alt algoritm cunoscut pentru determinarea arborelui parțial minim, tot de tip *Greedy*, este *algoritmul lui Prim*. Scrieți algoritmul şi programul corespunzătoare.

Problema 5. Codificarea Huffman



Presupunem că avem un text cu 262 de caractere şi ne dorim stocarea lui într-o formă cât mai redusă. Textul conține şase caractere (*p*, *q*, *r*, *x*, *y*, *z*), iar *p* apare de 100 de ori, *q* de 17 ori, *r* de două ori, *x* de 58 de ori, *y* de 80 de ori şi z de cinci ori.

Modul uzual de a reprezenta informațiile în calculator este cel folosind biții 0 şi 1. Ne interesează aşadar găsirea unei modalități de reprezentare a caracterelor folosind un *cod binar*, astfel încât fiecare

caracter este reprezentat cu ajutorul unei secvențe binare unice. În cazul nostru, dacă vom folosi un *cod binar de lungime fixă*, avem nevoie de minimum 3 biți pentru fiecare dintre cele șase caractere. Exemplu: $p = 000$, $q = 001$, $r = 010$, $x = 011$, $y = 100$, $z = 101$. În total, avem nevoie de $262 \cdot 3 = 786$ biți.

O alternativă la codul de lungime fixă este *codul de lungime variabilă*, care poate conduce la îmbunătățiri semnificative. Acest cod urmărește alocarea unor secvențe binare cât mai scurte caracterelor cel mai des folosite. Exemplu pentru textul nostru: $p = 0$, $q = 1101$, $r = 11000$, $x = 111$, $y = 10$, $z = 11001$. Numărul de biți necesari este: $100 \times 1 + 17 \times 4 + 2 \times 5 + 58 \times 3 + 80 \times 2 + 5 \times 5 = 537$, iar aceasta reprezintă 68,3% din 786, adică am obținut o reducere de circa 31,7% în raport cu codificarea de lungime fixă.

Codurile fix și variabil pentru un text de 262 de caractere, ce conține caracterele p, q, r, x, y, z, cu frecvențele indicate

	p	*q*	*r*	*x*	*y*	*z*	Total biți
Frecvență	100	17	2	58	80	5	–
Cod fix	000	001	010	011	100	101	786
Cod variabil	0	1101	11000	111	10	11001	537

Această codificare este validă, deoarece nici un cuvânt binar asociat unui caracter nu este prefix al cuvântului binar asociat altui caracter. O astfel de codificare se numește *codificare prefix* și simplifică atât codificarea (deci compactarea), cât și decodificarea. De exemplu, cuvântul binar 1011000110010111 se decodifică unic în *yrzpx*.

O astfel de codificare prefix a fost propusă de către David Albert Huffman (1925-1999) în septembrie 1952 (*Proceedings of the I.R.E.*, pp. 1098-1102): se folosește un arbore binar ale cărui frunze sunt caracterele date și interpretăm fiecare cod binar pentru un caracter ca fiind drumul de la rădăcină până la caracterul respectiv, unde 0 reprezintă deplasare pe fiul stâng și 1 deplasare pe fiul drept. Arborele ataşat unei codificări este un *arbore binar complet* (orice nod are zero sau doi fii) și nu este un arbore binar de căutare deoarece frunzele nu trebuie să fie ordonate și nodurile interne nu conțin chei pentru caractere.

Dacă notăm cu C alfabetul caracterelor, atunci arborele are $|C|$ frunze, câte unul pentru fiecare literă, și $|C| - 1$ noduri interne. Dacă notăm cu $f(c)$ frecvența unui caracter și cu $h(c)$ adâncimea frunzei în arbore (numărul de biți în codificare), atunci numărul total de caractere necesare este $\sum_{c \in C} f(c) \cdot h(c)$ și se mai numește *costul arborelui*.

Algoritmul lui Huffman este unul de tip *Greedy*: se începe cu o mulțime $|C|$ de frunze și se realizează $|C| - 1$ operații de fuzionare, în final obținându-se un singur arbore. La fiecare pas, se vor alege doi arbori de cost minim și se vor înlocui cu arborele obținut prin fuzionarea celor doi arbori. Prin fuzionarea a doi arbori, A_1 și A_2

...legem crearea unui nou arbore binar, în care cei doi arbori A_1 şi A_2 sunt fii stâng, ...spectiv drept ai rădăcinii.

Scrieți un program ce determină codificarea Huffman pentru un text dat, cu ...vențele caracterelor date.

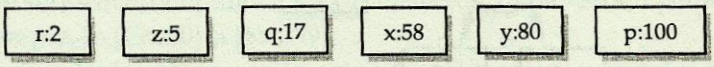
... de intrare. În fişierul *huffman.in* se găsesc perechile (caracter, frecvență), câte una ... linie. Caracterele pot fi litere ale alfabetului englez, iar frecvențele sunt numere ...rale între 1 şi 1000.

... de ieşire. Scrieți în fişierul *huffman.out* codificările pentru fiecare caracter, ca în ...plul:

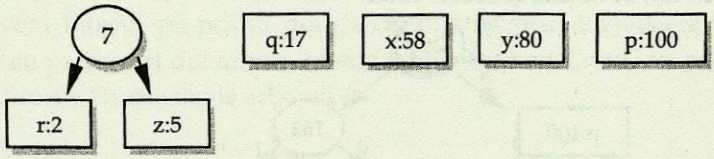
huffman.in	huffman.out
p 100	p 0
q 17	y 10
r 2	r 11000
z 58	z 11001
y 80	q 1101
x 5	x 111

...aliza problemei şi proiectarea soluției

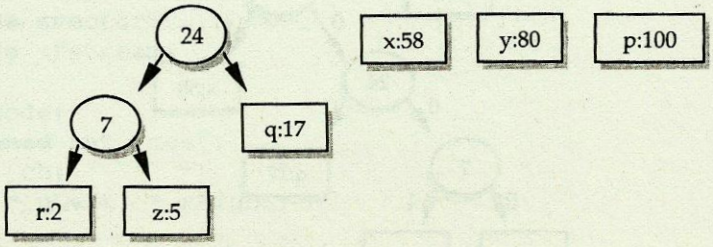
...voluția arborilor pentru exemplul dat este prezentată mai jos. Se începe cu şase ...oduri frunză:



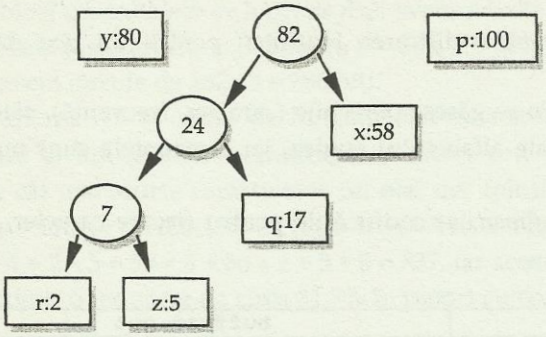
Se unesc arborii de cost minim şi se obțin următorii cinci arbori:



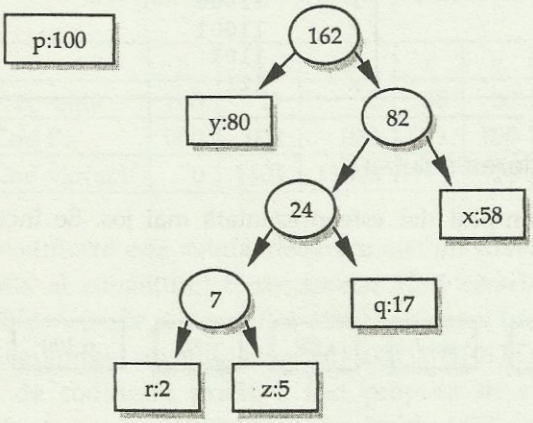
Prin fuzionarea arborilor de costuri 7 şi 17, se obțin:



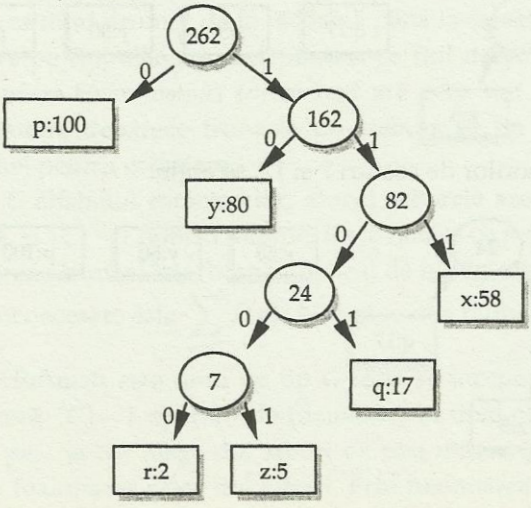
Arborii de costuri minime sunt cei cu 24 şi 58 în rădăcină:



La următorul pas, ultimii doi arbori au costurile 100 şi 162:



Prin unirea lor, se obţine arborele final:



care am etichetat muchiile cu valorile 0 şi 1. Acest arbore binar complet conţine codificarea Huffman, ea fiind pentru fiecare caracter şirul etichetărilor muchiilor drumului de la rădăcină la frunza corespunzătoare.

Algoritmul în pseudocod:

```
ALGORITM_Huffman(C)
  M ←ArboriFrunzăIniţiali(C)
  While (M are mai mult de un element) Do
    A1, A2 ←ArboriiCuCostMinim(M),
    Şterge A1, A2 din M
    A ←ArboreFuzionat(A1, A2)
    M.add(A)
  End_While
  return Codificări(M)
END_ ALGORITM_Huffman(C)
```

Complexitatea algoritmului este $O(n \log n)$, datorită operaţiei *căutare* a arborilor de cost minim. Algoritmul lui Huffman conduce la o codificare optimală, datorită proprietăţii de substructură optimă a subproblemelor. Demonstraţii pentru aceasta se găsesc, de exemplu, în [Cor00] şi [Knu99].

Vom scrie clasa *Node*, care va conţine ca atribute un caracter, costul şi cei doi arbori fii. Vom supraîncărca operatorul „<", deoarece ne propunem să salvăm mulţimea de arbori într-o structură de tip *std::multimap*, care îi va păstra ordonaţi şi la fiecare pas de procesare îi vom înlocui pe primii doi cu cel obţinut prin fuzionarea lor, folosind constructorul cu parametri doi arbori. Metoda membru *writeCosts()* va scrie în fişierul de ieşire codificările Huffman ale arborelui.

Program

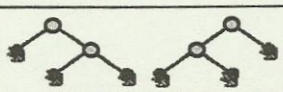
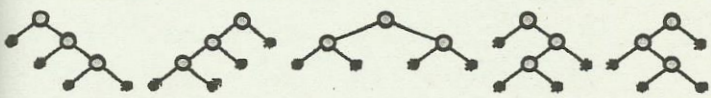
```
#include <map>
#include <vector>
#include <fstream>

class Node{
  unsigned int _cost;
  char _ch;
  Node *_nLeft, *_nRight;
public:
  Node(unsigned cost, char ch):
```

```
    _cost(cost), _ch(ch),_nLeft(NULL),_nRight(NULL){}
  Node(Node* pLeftChild,Node* pRightChild)
    :_cost(pLeftChild->_cost+pRightChild->_cost),
     _ch(0),_nLeft(pLeftChild),_nRight(pRightChild){}
  ~Node(){};
  inline unsigned int getCost()const {return _cost;}

  inline bool operator <(const Node &other) const{
    return _cost < other._cost;
  }
  void writeCosts(std::ofstream& out,
                  std::vector<bool> &rPathPrefix);
};

void Node::writeCosts(std::ofstream& out,
                  std::vector<bool> &rPathPrefix){
  if(_nLeft==NULL || _nRight==NULL ) {          //nod frunză
    out << _ch <<"\t";
    for(std::vector<bool>::const_iterator it=rPathPrefix.begin()
      ;it!=rPathPrefix.end();it++) {
        out << (*it ? 1:0);
    }
    out<<std::endl;
  }
  else {                                        //nod interior
    rPathPrefix.push_back(false);
    _nLeft->writeCosts(out,rPathPrefix);
    rPathPrefix[rPathPrefix.size()-1] = true;
    _nRight->writeCosts(out, rPathPrefix);
    rPathPrefix.pop_back();
  }
}

int main(){
  std::ifstream in("huffman.in");

  unsigned cost; char ch;
  std::multimap<int, Node*> s;
  while(in && !in.eof() && in>>ch>>cost){
    s.insert(std::pair<int, Node*>(cost, new Node(cost, ch)));
  }

  if(s.empty()) {
    return 0;
  }

  while(s.size()>1){
```

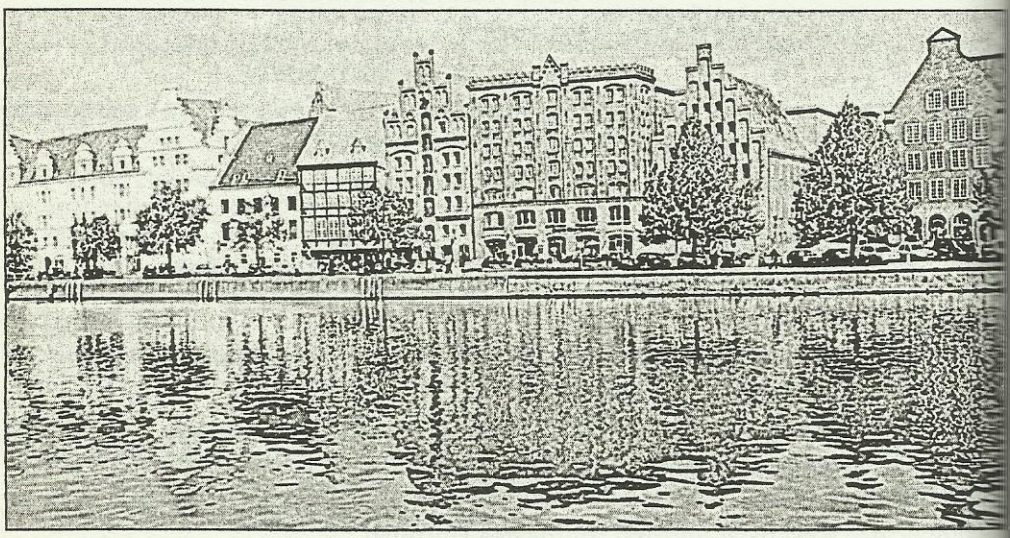
```
    std::multimap<int, Node*>::iterator it = s.begin();
    Node *pFirstNode = it->second;
    s.erase(it);
    it = s.begin();
    it = s.begin();
    Node *pSecondNode = it->second;
    s.erase(it);
    Node *pNewNode = new Node(pFirstNode, pSecondNode);
    s.insert(std::pair<int,Node*>(pNewNode->getCost(), pNewNode));
}

s.begin()->second->writeCosts(std::ofstream("huffman.out"),
                              std::vector<bool>());

return 0;
```

Probleme propuse

1. Desenați arborii Huffman pentru codificarea cuvintelor *BANANA, ABRACADABRA* și *MISSISSIPPI*.
2. Demonstrați că algoritmul Huffman conduce la o codificare optimală.
3. Scrieți o implementare C a algoritmului.
4. Extindeți programul astfel încât să furnizeze și procentul de reducere al codificării Huffman în raport cu codificarea de lungime fixă.
5. Modificați programul astfel încât la intrare să primească un fișier text și să se compacteze acest fișier folosind codificarea Huffman.
6. Demonstrați că numărul tuturor arborilor binari compleți cu n noduri interne este al n-lea număr din șirul lui Catalan: $C_n = \frac{1}{n+1} C_{2n}^n$.

	2 noduri interne
	3 noduri interne

Desenați cei 14 arbori binari compleți cu 4 noduri interne.



Case oglindite în Lübeck

Capitolul 7
Problema ordonării datelor

- Descrierea problemei
- Domeniul problemei. Definiții
- *POD* şi *PODI* sunt *NP*-complete
- Algoritmi pentru *POD* şi *PODI*
- Algoritmii *Random* (*RAN*)
- Algoritmii *Exact* (*EX*)
- Algoritmii *Greedy Min* (*GM*)
- Algoritmii *Greedy Min Simplified* (*GMS*)
- Algoritmii de mărginire inferioară (*LB*)
- Detalii de implementare
- Program (*STL*, parametri în linia de comandă)
- Interpretarea rezultatelor
- Probleme propuse

Dacă vrei să construieşti un vapor, să nu începi prin a-i trimite pe oameni după lemne, cuie, unelte, sfori şi alte materiale. Învaţă-i întâi să tânjească după marea îndepărtată, nesfârşită.
Antoine de Saint-Exupéry

Descrierea problemei

Se consideră o mulţime de cuvinte (mesaje) ce urmează să fie transmise pe un canal astfel încât ordinea de transmitere este irelevantă. Problema este determinarea secvenţei optime care minimizează numărul total de tranziţii pe canal.

În 1994, Stan şi Burleson au prezentat metoda *bus-invert*: cuvintele pot fi *inversate* şi apoi transmise pentru a reduce numărul total de transmisii ([Bur94]).

Prin *inversarea unui cuvânt* înţelegem operaţia prin care fiecare bit din cuvânt primeşte valoarea complementului său ($0 \to 1$, $1 \to 0$). Exemplu: pentru w = 10110001, $w_inversat$ = 01001110. Vom încerca în continuare să combinăm cele două paradigme într-una singură, adică vom permite cuvintelor să fie complementate, reordonate si apoi transmise. (Vom demonstra că această problemă – *Problema Ordonării Datelor cu Inversiune* [*PODI*] este NP-completă.) Aria de aplicaţii practice ale acestei probleme devine din ce în ce mai largă, de la testarea circuitelor până la transmiterea pachetelor în cadrul internetului şi de la *low-power design* până la implicaţii în biologie.

Domeniul problemei. Definiţii

Definiţia 1. Distanţa Hamming. Dacă cuvântul w_r este transmis urmat imediat de w_s, atunci numărul total de tranziţii este dat de numărul de biţi care se schimbă. Acesta este:

$$d(w_r, w_s) = \sum_{j=1}^{k} w_{rj} \oplus w_{sj}, \tag{1}$$

cunoscută, de asemenea, ca *distanţa Hamming* dintre w_r şi w_s. Aici, w_{rj} este al j-lea bit din reprezentarea lui w_r şi $\oplus$ este operaţia *SAU-EXCLUSIV*:

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

De exemplu, $d(10001110, 10110111) = 4$.

În cazul în care dispunem de mai multe cuvinte, reordonarea cuvintelor poate schimba semnificativ numărul total de tranziții.

În continuare, vom nota cu $\overline{w}$ complementul cuvântului w. Exemplu: $\overline{10110} = 01001$.

Propoziția 1. Fie k un număr natural pozitiv. Distanța Hamming definită pe spațiul cuvintelor de lungime k este o *metrică* pe spațiul total al cuvintelor, deoarece satisface următoarele:

1. $d(x,y) = 0 \Rightarrow x = y$ oricare ar fi două cuvinte de lungime k;
2. $d(x, y) = d(y, x)$ (simetrie);
3. $d(x,y) \leq d(x, z) + d(z, y)$ (inegalitatea triunghiului).

Demonstrație:

1. Din faptul că $d(x,y) = 0$, deducem:

$$\sum_{j=0}^{k} d(x_j, y_j) = 0 \quad \Rightarrow \quad d(x_j, y_j) = 0, \forall j = 1, k \quad \Rightarrow \quad x_j = y_j, \quad \forall j = 1, k \quad \Rightarrow \quad x = y \ (x_j \text{ şi } y_j$$

sunt biții din componența cuvintelor x şi y).

2. Egalitatea este evidentă datorită simetriei operației $\oplus$.

3. Presupunem că x, y şi z, cele trei cuvinte, au forma binară: $x_1...x_k$, $y_1...y_k$ şi $z_1...z_k$. Afirmația este adevărată pentru cuvinte de lungime 1, dacă vom considera biții a, b, c, atunci tabelul pentru expresiile $d(a, b)$ şi $d(a, c) + d(c, b)$ este:

Evaluarea expresiilor $d(a, b)$ şi $d(a, c) + d(c, b)$

a	b	c	$d(a, b)$	$d(a, c) + d(c, b)$
0	0	0	0	0
0	0	1	0	2
0	1	0	1	1
0	1	1	1	1
1	0	0	1	1
1	0	1	1	1
1	1	0	0	2
1	1	1	0	0

Din acest tabel observăm că inegalitatea triunghiului este satisfăcută pentru cuvinte de lungime 1 şi, deci, însumând biţii corespunzători, deducem că este adevărată pentru cuvinte de orice lungime. ❑

Definiţia 2. *Numărul total de tranziţii.* Considerându-se cuvintele w_1, w_2, ..., w_n cu semnificaţia de mai sus, o permutare δ (o funcţie bijectivă cu domeniul $\{w_1, w_2, ..., w_n\}$ şi codomeniul tot $\{w_1, w_2, ..., w_n\}$) a lor şi o asignare ρ (o funcţie cu domeniul $\{1, 2, ..., n\}$ şi codomeniul $\{0,1\}$) care specifică pentru fiecare cuvânt dacă se va considera complementat sau necomplementat, definim numărul total al tranziţiilor ca fiind valoarea:

$$N_T = \sum_{j=1}^{n-1} d(\delta(w_{\rho(j)}), \delta(w_{\rho(j+1)})) \tag{2}$$

Definiţia 3. *Matricea de adiacenţă.* Pentru o instanţă a problemei (unde n este numărul de cuvinte şi k lungimea fiecăruia), definim matricea de adiacenţă $A_{n \times n}$ unde $A(i, j) = d(w_i, w_j)$.

Exemplu: Considerăm $n = 3$, $k = 5$ şi cuvintele:

$w_1 = 00011$
$w_2 = 10110$
$w_3 = 01011$

$$A = \begin{pmatrix} 0 & 3 & 1 \\ 3 & 0 & 4 \\ 1 & 4 & 0 \end{pmatrix}$$

($d(w_1, w_1) = 0$, $d(w_1, w_2) = 3$, $d(w_1, w_3) = 1$, ...).

Referitor la această definiţie pentru matrice, deducem:

1. *Matricea este simetrică faţă de diagonala principală.* Datorită faptului că operaţia de SAU-EXCLUSIV este comutativă deducem că *distanţa Hamming* este comutativă, adică:

$$d(w_i, w_j) = d(w_j, w_i) \Rightarrow a_{ij} = a_{ji} \tag{4}$$

2. *Elementele de pe diagonala principală sunt nule.* Din definiţia distanţei Hamming deducem că:

$$d(w_i, w_i) = 0 \Rightarrow a_{ii} = 0 \tag{5}$$

3. Dacă inversăm toate cuvintele, matricea de adiacenţă este aceeaşi, ca şi în cazul când acestea sunt în forma iniţială:

$$d(w_i, w_j) = d(\overline{w_i}, \overline{w_j}) \tag{6}$$

4. Dacă lungimea cuvintelor este k, atunci este adevărată următoarea relație:

$$d(w_i, \overline{w_j}) = d(\overline{w_i}, w_j) = k - d(w_i, w_j) = k - a_{ij} \quad (7)$$

5. Dacă vom considera cuvintele în ordinea inițială, atunci numărul total de tranziții se poate obține însumând elementele de pe diagonalele vecine cu diagonala principală (de deasupra şi dedesubtul acesteia).

Exemplul 1. Considerăm $n = 3$, $k = 12$ şi următoarele cuvinte:

$w_1 = 010100111000$
$w_2 = 001010100110$
$w_3 = 001111011011$

$$A = \begin{pmatrix} 0 & 8 & 7 \\ 8 & 0 & 7 \\ 7 & 7 & 0 \end{pmatrix}$$

Dacă vom considera cuvintele în ordinea inițială şi neinversate, atunci numărul total de tranziții va fi:

$$N_T = a_{12} + a_{23} = 15$$

Dacă inversăm al doilea cuvânt, obținem:

$$N_T = (k - a_{12}) + (k - a_{23}) = (12 - 8) + (12 - 7) = 9,$$

ceea ce înseamnă că numărul total de tranziții s-a diminuat cu 40%.

Exemplul 2. Considerăm $n = 7$, $k = 15$ şi următoarele cuvinte:

$w_1 = 001010100110110$
$w_2 = 000000001111110$
$w_3 = 001111011010010$
$w_4 = 000010011110111$
$w_5 = 000001111011011$
$w_6 = 000110010001100$
$w_7 = 000001010001110$

$$A = \begin{pmatrix} 0 & 5 & 7 & 5 & 9 & 8 & 8 \\ 5 & 0 & 8 & 4 & 6 & 7 & 5 \\ 7 & 8 & 0 & 6 & 6 & 7 & 7 \\ 5 & 4 & 6 & 0 & 6 & 7 & 7 \\ 9 & 6 & 6 & 6 & 0 & 9 & 5 \\ 8 & 7 & 7 & 7 & 9 & 0 & 4 \\ 8 & 5 & 7 & 7 & 5 & 4 & 0 \end{pmatrix}$$

Dacă vom considera cuvintele în ordinea inițială şi neinversate, numărul total de tranziții va fi:

$$N_T = \sum_{t=1}^{6} a_{t,t+1} = 38$$

Dacă vom schimba doar ordinea cuvintelor w_1, w_2, w_4, w_3, w_5, w_7, w_6 şi le vom păstra neinversate, obţinem:

$$N_T = 30$$

ce ar însemna o reducere de 26,67%.

Exemplul 3. Considerăm $n = 8$, $k = 13$ şi următoarele cuvinte:

w_1 = 1111010111001
w_2 = 1010100110110
w_3 = 0000001111110
w_4 = 1111011010010
w_5 = 0001111011011
w_6 = 0110010001100
w_7 = 0001010001110
w_8 = 1000110110100

$$A = \begin{pmatrix} 0 & 8 & 9 & 5 & 7 & 6 & 8 & 7 \\ 8 & 0 & 5 & 7 & 9 & 8 & 8 & 3 \\ 9 & 5 & 0 & 8 & 6 & 7 & 5 & 6 \\ 5 & 7 & 8 & 0 & 6 & 7 & 7 & 8 \\ 7 & 9 & 6 & 6 & 0 & 9 & 5 & 8 \\ 6 & 8 & 7 & 7 & 9 & 0 & 4 & 7 \\ 8 & 8 & 5 & 7 & 5 & 4 & 0 & 7 \\ 7 & 3 & 6 & 8 & 8 & 7 & 7 & 0 \end{pmatrix}$$

Dacă vom considera cuvintele în ordinea iniţială şi neinversate, numărul total de tranziţii va fi:

$$N_T = \sum_{i=1}^{7} a_{i,i+1} = 47$$

Dacă vom considera cuvintele reordonate şi unele inversate astfel:

$$w_1 \to \overline{w}_3 \to w_4 \to \overline{w}_8 \to \overline{w}_2 \to w_5 \to w_6 \to w_7$$

obţinem următorul număr de tranziţii:

$$N_T = (13 - a_{13}) + (13 - a_{34}) + (13 - a_{48}) + a_{82} + (13 - a_{25}) + a_{56} + a_{67} = 29$$

iar reducerea obţinută este de 38,30%.

Din aceste exemple, observăm că reordonarea şi inversarea anumitor cuvinte pot duce la reduceri mari de resurse. Dacă pentru aceste cazuri, în care dimensiunile n şi k sunt foarte mici, folosirea unui algoritm determinist pentru aflarea optimului este posibilă, atunci când valorile cresc, un astfel de algoritm nu are utilitate practică. Soluţia ar fi căutarea unor metode care să ne furnizeze valori cât mai apropiate de valorile optime.

Definiţia 4. POD (Problema ordonării datelor). Găsiţi o permutare σ a şirului de cuvinte $w_1, w_2, \ldots, w_n$, astfel încât numărul total de tranziţii

$$N_T = \sum_{j=1}^{n-1} d(w_{\sigma(j)}, w_{\sigma(j+1)}) \quad (8)$$

să fie minimizat.

Definiţia 5. PODI (Problema ordonării datelor cu inversiune). Găsiţi o permutare σ a şirului de cuvinte $w_1, w_2, \ldots, w_n$ şi o asignare δ astfel încât numărul total de tranziţii

$$N_T = \sum_{j=1}^{n-1} d(\delta(w_{\sigma(j)}), \delta(w_{\sigma(j+1)})) \quad (9)$$

să fie minimizat.

POD şi *PODI* sunt *NP*-complete

În [Mur95] se demonstrează că *POD* este *NP*-completă şi au fost propuse o serie de scheme pentru găsirea unor soluţii acceptabile: arbore parţial dublu (*Double Spanning Tree – DST*), arbore parţial minim cu potrivire maximă (*Minimum Spanning Tree – Maximum Matching – ST-MM*) şi *Greedy* (care din punct de vedere empiric a fost găsită drept cea mai bună).

Clasa *NP* reprezintă cea mai interesantă clasă de probleme. Una dintre explicaţii ar fi aceea că, dacă se găseşte un algoritm polinomial ce rezolvă o problemă *NP*-completă, atunci există câte un algoritm polinomial pentru rezolvarea oricărei probleme *NP*-complete. Datorită vastei arii pe care o acoperă problemele *NP*-complete studiate până în prezent fără să se fi făcut vreun mic pas în direcţia rezolvării vreuneia dintre ele, ar fi uluitor să se descopere vreodată rezolvarea unei astfel de probleme în timp polinomial.

Pentru a afirma că o problemă este *NP*-completă, trebuie furnizată o demonstraţie a inaccesibilităţii sale. Pentru un inginer, de exemplu, este deseori mai indicat dacă, relativ la o problemă din această clasă, încearcă să găsească un algoritm de aproximare decât să caute un algoritm exact. Există un număr foarte mare de probleme obişnuite şi interesante, care la prima vedere nu par să fie mai dificile, decât o sortare sau o căutare în grafuri şi care, totuşi, sunt *NP*-complete. Problemele *NP*-complete apar în cele mai diverse domenii: logică, grafuri, aritmetică, proiectarea reţelelor, mulţimi şi partiţii, memorări şi căutări, planificări, programare matematică, algebră şi teoria numerelor, jocuri, teoria limbajelor şi a automatelor, optimizarea programelor etc.

PODI se transformă într-o problemă de decizie după logica următoare.

Instanța. Se consideră o mulțime de n cuvinte $w_1, w_2, ..., w_n$, fiecare de dimensiune k: $w_i \in \{0, 1\}^k$ și un număr pozitiv M.
Întrebare. Există permutare σ a șirului de cuvinte $w_1, w_2, ..., w_n$ și o asignare δ a celor n cuvinte, astfel încât

$$\sum_{j=1}^{n-1} d(\delta(w_{\sigma(j)}), \delta(w_{\sigma(j+1)})) \leq M \; ? \qquad (10)$$

$\delta(w_i) = w_i$ sau $\delta(w_i) = \overline{w_i}$.

Știm că *POD* este *NP*-completă și, intuitiv, se deduce că și *PODI* este cel puțin tot atât de grea, deoarece are un grad mai mare de libertate privind plasarea obiectelor.

Teoremă. PODI este *NP*-completă.
Demonstrație. Vom transforma *POD* în *PODI*. *POD* are ca problemă de decizie următoarea formă:
Instanța. Se consideră o mulțime de N cuvinte $W_1, W_2, ..., W_N$, fiecare de dimensiune K: $W_i \in \{0, 1\}^K$, și un număr pozitiv L.
Întrebare. Există permutare σ a celor N cuvinte date, astfel încât

$$\sum_{j=1}^{n-1} d(w_{\sigma(j)}, w_{\sigma(j+1)}) \leq L \; ? \qquad (11)$$

Considerându-se o instanță pentru *POD*, generăm următoarea instanță pentru *PODI*. Fie $n = N$ și $M = L$. Pentru fiecare i, construim w_i concatenând KN zerouri după W_i, adică $w_i = W_i00...0$ (KN zerouri). Astfel, $k = K+KN$ și timpul de transformare este polinomial. Atunci există o permutare σ a cuvintelor W_i, astfel încât relația (11) este satisfăcută dacă și numai dacă există o permutare a cuvintelor w_i, astfel încât relația (10) este satisfăcută. ❑

Algoritmi pentru *POD* și *PODI*

Presupunem că avem la dispoziție tipurile abstracte *Permutation* și *BitString*, care permit manipularea permutărilor, respectiv a șirurilor de biți. Vom descrie în continuare câțiva algoritmi în pseudocod pentru *PODI*, problema fiind o extindere a problemei *POD*. Pentru o instanță a problemei, adică n cuvinte de lungime k, *PODI* necesită returnarea unei permutări de ordin n (reprezentând ordinea de transmitere a cuvintelor) și a unui șir de biți tot de lungime n (reprezentând asignările cuvintelor). Algoritmii descriși vor fi *Random POD* și *PODI*, *Exact POD* și *PODI*, două metode *Greedy* pentru ambele probleme și un algoritm de mărginire inferioară (engl. *lower-bound*), folosind arborele parțial minim de acoperire.

Considerăm $n = 8$, $k = 9$ şi cuvintele:

$w_1 = 010110011$
$w_2 = 101110000$
$w_3 = 001000001$
$w_4 = 111010001$
$w_5 = 111011111$
$w_6 = 000101111$
$w_7 = 100010001$
$w_8 = 001001111$

Matricea de adiacenţă $A_89 =$

$$\begin{pmatrix} 0 & 5 & 5 & 4 & 5 & 4 & 4 & 6 \\ 5 & 0 & 4 & 3 & 6 & 7 & 3 & 7 \\ 5 & 4 & 0 & 3 & 6 & 5 & 3 & 3 \\ 4 & 3 & 3 & 0 & 3 & 8 & 2 & 6 \\ 5 & 6 & 6 & 3 & 0 & 5 & 5 & 3 \\ 4 & 7 & 5 & 8 & 5 & 0 & 6 & 2 \\ 4 & 3 & 3 & 2 & 5 & 6 & 0 & 6 \\ 6 & 7 & 3 & 6 & 3 & 2 & 6 & 0 \end{pmatrix}$$

Vom numi această instanţă a problemei *EX89* şi ne vom referi la ea pentru a exemplifica soluţiile algoritmilor prezentaţi în continuare. Costul total pentru cazul când cuvintele sunt transmise în ordinea iniţială şi neinversate este 32.

Algoritmii *Random* (*RAN*)

Algoritmul *Random* pentru *POD* va genera o permutare aleatorie şi o va returna împreună cu costul ataşat. Cel pentru *PODI* va genera o permutare şi un şir de biţi reprezentând asignarea.

```
ALGORITM_ RAN_POD(n, k, w1...wn)
        p ← random Permutation
        return p, cost(p)
END_ ALGORITM_ RAN_POD(n, k, w1...wn)
```

```
ALGORITM_RAN_PODI(n, k, w1...wn)
        p ← random Permutation
        b ← random BitString
        return p, b, cost(p, b)
END_ ALGORITM_RAN_PODI(n, k, w1...wn)
```

De exemplu, pentru permutarea aleatorie (7, 8, 4, 2, 3, 5, 6, 1), costul *POD* pentru ... este 34. Pentru permutarea (7, 1, 3, 8, 5, 2, 4, 6) şi asignarea (0, 1, 0, 1, 1, 1, 0, 0), ... *PODI* este 38.

Algoritmii *Exact* (*EX*)

Algoritmul *Exact* pentru *POD* generează toate permutările şi o reţine pe prima, care conduce la un cost minim. Cel pentru *PODI* generează toate perechile posibile permutare-asignare şi o reţin pe prima cu costul minim.

Algoritmul *Exact* pentru *POD* va genera o permutare aleatorie şi o va returna, împreună cu costul ataşat. Cel pentru *PODI* va genera o permutare şi un şir de biţi reprezentând asignarea.

```
ALGORITM_ EXACT_POD(n, k, w1...wn)
        p ← random Permutation(n)
        For (all Permutations(n) r) Execute
          If(cost(r)<cost(p)) p ← r
        End_For
        return p, cost(p)
END_ ALGORITM_ EXACT_POD(n, k, w1...wn)
```

```
ALGORITM_ EXACT_PODI(n, k, w1...wn)
        p ← random Permutation(n)
        b ← random BitString(n)
        For (all Permutations(n) r) Execute
          For (all BitStrings(n) bs) Execute
            If(cost(r, bs)<cost(p, b)) Then
              p ← r
              b ← bs
        End_If  End_For  End_For
        return p, b, cost(p, b)
END_ ALGORITM_ EXACT_PODI(n, k, w1...wn)
```

Pentru exemplul *EX89*, costul minim *POD* este 21 şi se obţine cu permutarea (1, 8, 3, 2, 7, 4, 5). Costul minim *PODI* este 16 şi se obţine cu permutarea (1, 8, 2, 6, 4, 7, 5), iar asignarea este (0, 1, 0, 1, 0, 0, 0, 1). Complexitatea algoritmului *EXACT_POD* este $O(n!)$; pentru *EXACT_PODI* este $O(n!\cdot 2^n)$.

Algoritmii *Greedy Min* (*GM*)

Greedy Min funcţionează pentru ambele probleme, *POD* şi *PODI*. Ei funcţionează după cum urmează:

- calculează costul pentru toate perechile de cuvinte distincte şi selectează perechea cu cost minim;
- această pereche cu cost minim este secvenţa de început, formată din două cuvinte;
- construieşte progresiv secvenţa, adăugând la fiecare pas cel mai convenabil cuvânt care nu a fost încă adăugat. Acest cuvânt poate fi adăugat la sfârşitul sau la începutul secvenţei, după cum costul obţinut este minimal. În cazul *PODI* se adaugă, bineînţeles, şi libertatea de inversare a cuvântului adăugat.

Complexitatea algoritmului este polinomială $O(n^2)$ pentru ambele probleme. Pentru exemplul *EX89*, permutarea obţinută pentru *POD* este (5, 1, 6, 8, 3, 4, 7, 2) şi conduce la costul 22. Pentru *PODI* se obţin permutarea (5, 3, 7, 4, 6, 2, 8, 1) şi asignarea (1, 0, 0, 0, 1, 0, 1, 0) şi conduc la un cost total 16.

Algoritmii *Greedy Min Simplified* (*GMS*)

Aceştia funcţionează, de asemenea, pentru ambele probleme, *POD* şi *PODI*. Diferenţa este că, la fiecare pas, adăugarea noului cuvânt (cu posibilitatea de a se seta şi inversa, în cazul *PODI*) este permisă doar la un capăt, al secvenţei. Explicaţia este că se economiseşte timp prin testarea numai la un capăt şi performanţele în practică pentru mulţimi mari de date sunt foarte apropiate de *Greedy Min*. Pentru operaţia de transmitere a cuvintelor (prin intermediul internetului, de exemplu), timpul este un factor important. Complexitatea este, de asemenea, polinomială $O(n^2)$.

Pentru exemplul *EX89*, permutarea obţinută pentru *POD* este (4, 7, 2, 3, 8, 6, 1, 5) şi conduce la costul total 23. Pentru cazul *PODI*, se obţin permutarea (4, 6, 8, 1, 2, 7, 3, 5) şi asignarea (0, 1, 1, 0, 1, 1, 1, 0), care conduc la costul 19.

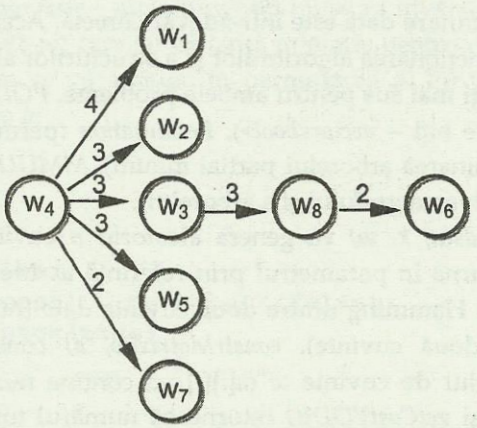
Algoritmii de mărginire inferioară (*LB*)

Problemele *POD* şi *PODI* se pot modela natural cu ajutorul teoriei grafurilor. Pentru *POD* se construieşte graful complet cu vârfurile reprezentând cele n cuvinte şi muchiile având drept costuri distanţa dintre cuvintele reprezentând capetele sale. Conform secţiunii teoretice de mai sus, distanţa dintre două cuvinte este aceeaşi dacă

două sunt în aceeaşi stare de inversare (ambele inversate sau ambele în forma ală). De asemenea, dacă ele sunt în stări diferite, distanța este aceeaşi pentru ele posibilități. În cazul *PODI*, se va construi un multigraf: între două vârfuri te putem avea şi două muchii, reprezentând amândouă posibilitățile (ambele şi fază sau faze diferite), atunci când cele două costuri rezultate sunt diferite. emele *POD* şi *PODI* cer determinarea unor drumuri hamiltoniene de cost minim f, respectiv multigraf, care sunt probleme *NP*-complete. Un arbore minimal de perire pe acest graf (respectiv multigraf) va avea un cost mai mic decât drumul iltonian de cost minim. Nu putem folosi un astfel de arbore ca soluție pentru ema dată, dar rezultatul se poate utiliza pentru a măsura deviațiile de la minim zate de algoritmii *greedy* prezentați mai sus.

Pentru determinarea arborelui parțial de cost minim există doi algoritmi: *Prim* loseşte vârfurile) şi *Kruskal* (foloseşte muchiile). Vom implementa algoritmul skal, prezentat în capitolul precedent, pentru a furniza limite inferioare ale urilor.

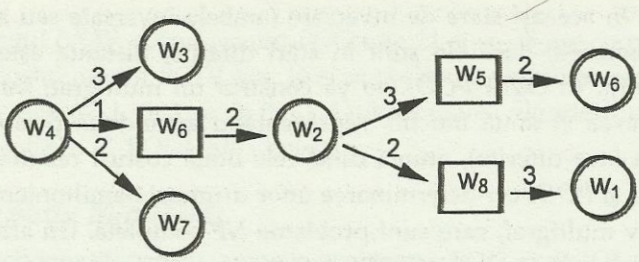
Pentru *POD*, exemplul *EX89*, se va obține arborele din figură ((4, 7) → 2 | (6, 8) → (2, 4) → 3 | (3, 4) → 3 | (3, 8) → 3 | (4, 5) → 3 | (1, 4) → 4).



Arbore parțial de cost minim pentru POD, EX89

Costul obținut este aşadar 20.

Pentru *PODI* se va obține arborele: (4, 6) → 1 | (2, 6) → 2 | (2, 8) → 2 | (4, 7) → 2 | 8) → 3 | (2, 5) → 3 | (3, 4) → 3, care are costul 16.



Arbore parţial de cost minim pentru PODI, EX89

Complexitatea algoritmului este polinomială $O(n^2)$.

Detalii de implementare

Ca de obicei, vom lăsa deoparte în cadrul metodelor din program validarea corectitudinii datelor de intrare. Vom presupune că toate sunt corecte şi nu vom testa, de exemplu, dacă există cuvinte de lungimi diferite, dacă avem mai puţin de două cuvinte sau dacă o permutare dată este într-adevăr corectă. Acest lucru ne va permite să ne concentrăm pe funcţionarea algoritmilor şi a structurilor alese. Vom implementa toţi algoritmii prezentaţi mai sus pentru ambele probleme, *POD* şi *PODI*. Vom defini tipurile *BitString* (şir de biţi – *vector<bool>*), *Permutation* (permutare), *TPair* (perechi necesare pentru determinarea arborelui parţial minim), *MMUU* (*multimap*, tot pentru algoritmul Kruskal – salvarea grafului şi a arborelui).

Metoda *generateWords(n, k, w)* va genera aleatoriu *n* cuvinte binare, fiecare de lungime *k*, şi le va returna în parametrul prin referinţă *w*. Metoda *hammingDist(w1, w2)* returnează distanţa Hamming dintre două cuvinte date (numărul de biţi corelaţi care coincid în cele două cuvinte). *constrMatrix(w, a)* construieşte matricea de adiacenţă pe baza şirului de cuvinte *w* (*a[i][j]* va conţine numărul de biţi corelaţi diferiţi). *getCostPOD()* şi *getCostPODI()* returnează numărul total de tranziţii pentru instanţe ale celor două probleme (şir de cuvinte, matricea de adiacenţă, permutare, pentru *PODI*, adiţional, şi o asignare *bs*).

Metodele *getRandomCostPOD()* şi *getRandomCostPODI()* furnizează aleatoriu soluţii ale celor două probleme şi returnează costurile asociate lor. Remarcaţi utilizarea algoritmului *random_shuffle()* din *header*-ul *<algorithm>*, care amestecă elementele unui container situate între iteratorii daţi ca parametru.

getExactPOD() determină o soluţie optimă pentru problema *POD* şi o returnează în al treilea parametru – permutarea *p*. Iniţial, *p* este permutarea identică, toate permutările se vor genera lexicografic şi permutarea optimă va fi mereu actualizată. Remarcaţi utilizarea algoritmului *next_permutation()*, care generează următoarea

permutare în ordine lexicografică a permutării reprezentate de iteratorii-parametru. Metoda returnează costul optimal. *getExactPODI()* determină soluția optimă pentru o instanță a *PODI* și o returnează în ultimii doi parametri – permutarea *p*, asignarea *bs*. Pentru fiecare permutare se vor genera toate asignările posibile (de le 000...00 la 111...11) și se va reține optimul actual. Pentru a determina succesorul unui șir de biți, scriem funcția *next_bitstring()*, care transformă parametrul în succesorul său, dacă acesta există, caz în care returnează *true*. Exemplu: *succesor*(110111011) = 110111100. Dacă succesorul nu există, *next_bitstring()* returnează *false*. Succesorul unui șir de biți se obține prin transformarea în 1 a primului bit 0, considerat de la dreapta spre stânga și transformarea în 0 a tuturor biților succesori (toți sunt 1!).

Metoda *getLB()* returnează limita inferioară a unei instanțe și se poate aplica ambelor probleme prin setarea corespunzătoare a celui de-al treilea parametru *isPODI*. Dacă se cere rezolvarea *PODI*, atunci se vor introduce în multigraful inițial și costurile muchiilor pentru cuvintele cu asignări diferite ($k - a[i][j]$). Implementarea este foarte asemănătoare cu cea prezentată detaliat în capitolul 6, *Greedy*, la problema 4 (Kruskal).

getGMS() rezolvă problemele *POD* și *PODI* folosind algoritmul *Greedy Min Simplified*, prezentat mai sus. Dacă se cere rezolvarea *POD*, atunci vectorul *b* va conține numai elemente *false* – nici un cuvânt nu se va inversa. În continuare, se caută perechea de cuvinte (*i0*, *j0*) care au distanță minimă, pentru cazul *PODI* se verifică și valorile $k - a[i][j]$, care se va adăuga în permutarea *p*. Pentru cazul *PODI*, trebuie actualizată și asignarea *b*:

```
p.push_back(i0);
p.push_back(j0);
if(isPODI){
  b.push_back(false);
  if(a[i0][j0]==cost) b.push_back(false);
    else b.push_back(true);
}
```

Cât timp permutarea *p* curentă nu are încă *n* elemente ($p.size() < n$), se caută următorul cuvânt care va fi introdus. Acest cuvânt (indice *next*) nu trebuie să se afle încă în *p* ($find(p.begin(), p.end(), i) == p.end()$) și să formeze cost minim cu ultimul element din *p* (*j0*). Remarcați folosirea algoritmului *find()* pus la dispoziție de *STL*. Pentru cazul *PODI*, se verifică și valoarea $k - a[j0][i]$. Valoarea *next* găsită se va adăuga lui *p* și, în cazul *PODI*, se va actualiza și asignarea *b* în concordanță cu ultimul element introdus în *p* și ultimul element din *b*.

Metoda *Greedy Min* este implementată prin *getGM()*. Deoarece ea permite operarea la ambele capete ale secvenței curente, vom folosi listele auxiliare *pAux* și *bAux* pentru a reprezenta permutarea și asignarea. Vom reține de această dată ambele capete ale

secvenței în *i0* și *j0*, la fiecare pas se va adăuga cuvântul cel mai convenabil, încă neadăugat, ținând cont și dacă se cere rezolvarea *PODI*. La sfârșit, cele două liste se vor copia în *p* și *b* prin:

```
for(size_t i=0; i<n; i++){
    p.push_back(pAux.front()); pAux.pop_front();
    if(isPODI){
      b.push_back(bAux.front());
      bAux.pop_front();
    }
}
```

Remarcați utilizarea metodelor *pop_front()*, *push_front()*, *front()*, caracteristice listelor.

Ne propunem aplicarea tuturor algoritmilor implementați pentru diferite instanțe aleatorii ale problemei. Pentru aceasta vom folosi șase argumente în linia de comandă: *n1*, *n2*, *sn*, *k1*, *k2*, *sk*, cu semnificația că se vor genera și analiza succesiv instanțe pentru valori ale lui *n* între *n1* și *n2*, cu pasul *sn*, și pentru fiecare valori ale lui *k* între *k1* și *k2*, cu pasul *sk*:

```
for(n=n1; n<=n2; n+=sn)
  for(k=k1; k<=k2; k+=sk)... // aplică toți algoritmii ...
```

Dacă numele executabilului este *podpodi.exe*, atunci se va executa comanda:

```
podpodi 5 9 1 25 65 20
```

Rezultatele se vor scrie în fișierul *report.txt*, în următoarea formă:

		POD					PODI				
n	**k**	**RAN**	**EX**	**GMS**	**GM**	**LB**	**RAN**	**EX**	**GMS**	**GM**	**LB**
5	25	48	40	41	40	37	61	35	38	35	34
5	45	90	79	84	79	79	91	76	79	76	76
5	65	134	117	120	117	117	134	116	118	116	115
6	25	66	55	55	55	55	58	47	47	47	47
6	45	114	95	98	95	95	121	86	88	86	84
6	65	158	146	155	146	142	167	136	139	136	136
...											

Urmăriți implementarea în programul principal. Deoarece ne propunem apelul tuturor algoritmilor, deci și a celor exacți, timpul de execuție crește considerabil. De aceea, valorile pentru *n* și *k* trebuie să fie reduse.

Program (*STL*, parametri în linia de comandă)

```
#include <vector>
#include <algorithm>
#include <map>
#include <list>
#include <fstream>

using namespace std;

typedef vector<bool> BitString;
typedef vector<unsigned> Permutation;

typedef pair<unsigned, unsigned> TPair;
typedef multimap<unsigned, unsigned> MMUU;

bool generateWords(unsigned n, unsigned k,
                   vector<BitString>& w){
  for(unsigned i=0; i<n; i++){
    BitString wAux; short moneda;
    for(unsigned j=0; j<k; j++){
      moneda = (int)(2*((double)rand()/(RAND_MAX+1)));
      wAux.push_back(moneda ? true : false);
    }
    w.push_back(wAux);
  }
  return true;


int hammingDist(BitString w1, BitString w2){
  if(w1.size() != w2.size()) return -1;
  int t=0;
  for(size_t i=0; i<w1.size(); i++)
    if(w1[i]!=w2[i]) t++;
  return t;


void constructMatrix(vector<BitString>& w,
                     vector<vector<unsigned> >& a){
  unsigned n = w.size();
  unsigned k = w[0].size();
  for(unsigned i=0; i<n; i++)
    for(unsigned j=0; j<n; j++)
      a[i][j] = hammingDist(w[i], w[j]);
```

```
unsigned getCostPOD(vector<BitString>& w,
                    vector<vector<unsigned> >& a,
                    Permutation& p){
  unsigned t = 0;
  for(size_t i = 1; i<w.size(); i++)
    t += a[p[i-1]][p[i]];
  return t;
}

unsigned getCostPODI(unsigned n, unsigned k,
                     vector<vector<unsigned> >& a,
                     Permutation& p, BitString& bs){
  unsigned t = 0;
  if(n <=1 || p.size()!=n || bs.size()!=n)
             return 0;
  for(unsigned i = 1; i<n; i++)
    t += (bs[i]==bs[i-1])?a[p[i-1]][p[i]]:k-a[p[i-1]][p[i]];
  return t;
}

unsigned getRandomCostPOD(unsigned n, unsigned k,
                          vector<vector<unsigned> >& a,
                          Permutation& p){
  for(unsigned i=0; i<n; i++)p.push_back(i);
  random_shuffle(p.begin(), p.end());
  return getCostPOD(n, k, a, p);
}

unsigned getRandomCostPODI(unsigned n, unsigned k,
                           vector<vector<unsigned> >& a,
                       Permutation& p, BitString& b){

  p.clear(); b.clear();
  for(unsigned i=0; i<n; i++){
     p.push_back(i);
     b.push_back((int)(2*((double)rand()/(RAND_MAX+1))));
  }
  random_shuffle(p.begin(), p.end());
  return
    getCostPODI(n, k, a, p, b);
}


unsigned getExactPOD(unsigned n, unsigned k,
                     vector<vector<unsigned> >& a,
                     Permutation& p){
  unsigned t=getRandomCostPOD(n, k, a, p);
```

```
  Permutation r;
  p.clear();
  for(unsigned i=0; i < n; i++) r.push_back(i);
  while(next_permutation(r.begin(), r.end())){
    unsigned aux = getCostPOD(n, k, a, r);
    if(aux<t){
      t=aux;
      p=r;
    }
  }
  return t;


bool next_bitstring(BitString& bs){
  int idx = bs.size()-1;
  while(idx>=0 && bs.at(idx)){
    bs[idx--]=false;
  }
  if(idx>=0){
    bs[idx] = true;
    return true;
  }
  return false;


unsigned getExactPODI(unsigned n, unsigned k,
                      vector<vector<unsigned> >& a,
                      Permutation& p, BitString& bs){
  unsigned t=getRandomCostPODI(n, k, a, p, bs);
  Permutation r;
  p.clear();
  for(unsigned i=0; i < n; i++) r.push_back(i);
  while(next_permutation(r.begin(), r.end())){
    BitString bs2;
    for(unsigned i=0; i < n; i++)bs2.push_back(false);
    do{
      unsigned aux = getCostPODI(n, k, a, r, bs2);
      if(aux<t){
        t=aux;
        p=r;
        bs=bs2;
      }
    } while(next_bitstring(bs2));
  }
  return t;
}
```

```
unsigned getLB(unsigned n, unsigned k,
               vector<vector<unsigned> >& a,
               bool isPODI, MMUU& H){
  H.clear();
  MMUU E;
  for(unsigned i=0; i<n-1; i++)
    for(unsigned j=0; j<n; j++){
      E.insert(TPair(a[i][j], i*n+j));
      if (isPODI) E.insert(TPair(k-a[i][j], i*n+j));
  }
  vector<unsigned> C;
  unsigned cost = 0;
  for(unsigned i=0; i<n; i++) C.push_back(i);
  while(H.size()<n-1){
    unsigned uu, vv;
    MMUU::iterator it = E.begin();
    uu = (it->second)/n; vv = (it->second)%n;
    while(C[uu]==C[vv]){
      it++;
      uu = it->second/n;
      vv = it->second%n;
    };
    H.insert(TPair(it->first, it->second));
    cost += it->first;
    E.erase(it);
    unsigned mi = min(C[uu], C[vv]);
    unsigned ma = max(C[uu], C[vv]);
    for(unsigned i=0; i<n; i++)
      if(C[i]==ma) C[i]=mi;

  }
  return cost;
}

unsigned getGMS(unsigned n, unsigned k,
                vector<vector<unsigned> >& a,
                bool isPODI,
                Permutation& p, BitString& b){
  p.clear(); b.clear();
  if(!isPODI)
    for(unsigned i=0; i<n; i++) b.push_back(false);
  unsigned i0=0, j0=1;
  unsigned cost = a[i0][j0];
  for(unsigned i=0; i<n-1; i++)
    for(unsigned j=i+1; j<n; j++){
      if(a[i][j]<cost){
        i0=i; j0=j;
```

```
      cost = a[i0][j0];
    };
    if(isPODI && k-a[i][j]<cost){
      i0=i; j0=j;
      cost = k-a[i][j];
    }
  }
  p.push_back(i0);
  p.push_back(j0);
  if(isPODI){
    b.push_back(false);
    if(a[i0][j0]==cost) b.push_back(false);
    else b.push_back(true);
  }
  while(p.size()<n){
    unsigned next, aux = k+1;
    for(unsigned i=0; i<n; i++){
      if(find(p.begin(), p.end(), i)==p.end()&&a[j0][i]<aux){
        if(a[j0][i]<aux){
          aux = a[j0][i]; next = i;
        }
        if(isPODI && k-a[j0][i]<aux){
          aux = k-a[j0][i]; next = i;
        }
      }
    }
    p.push_back(next);
    cost += aux;
    if(isPODI){
      bool bLast = b[b.size()-1];
      if(aux==a[j0][next]) b.push_back(bLast);
      else b.push_back(!bLast);
    }
    j0=next;
  }
  return cost;

unsigned getGM(unsigned n, unsigned k,
               vector<vector<unsigned> >& a,
               bool isPODI,
               Permutation& p, BitString& b){
  p.clear(); b.clear();
  list<unsigned> pAux;
  list<bool> bAux;
  if(!isPODI)
    for(unsigned i=0; i<n; i++) b.push_back(false);
  unsigned i0=0, j0=1;
```

```
  unsigned cost = a[i0][j0];
  for(unsigned i=0; i<n-1; i++)
    for(unsigned j=i+1; j<n; j++){
      if(a[i][j]<cost){
        i0=i; j0=j;
        cost = a[i0][j0];
      };
      if(isPODI && k-a[i][j]<cost){
        i0=i; j0=j;
        cost = k-a[i][j];
      }
    }
  pAux.push_back(i0);
  pAux.push_back(j0);
  if(isPODI){
    bAux.push_back(false);
    if(a[i0][j0]==cost) bAux.push_back(false);
    else bAux.push_back(true);
  }
 while(pAux.size()<n){
  unsigned next, aux = k+1;
  for(unsigned i=0; i<n; i++){
    if(find(pAux.begin(), pAux.end(), i)==pAux.end()){
      if(a[j0][i]<aux){
        aux = a[j0][i]; next = i;
      };
      if(a[i0][i]<aux){
        aux = a[i0][i]; next = i;
      };

      if(isPODI && k-a[j0][i]<aux){
        aux = k-a[j0][i]; next = i;
      };
      if(isPODI && k-a[i0][i]<aux){
        aux = k-a[i0][i]; next = i;
      }
    }
  }

  cost += aux;
  if(a[j0][next]==aux || (isPODI && k-a[j0][next]==aux)){
    pAux.push_back(next);
    if(isPODI){
      bool bLast = bAux.back();
```

```
      if(aux==a[j0][next]) bAux.push_back(bLast);
      else bAux.push_back(!bLast);
    }
    j0 = next;
  } else {
    pAux.push_front(next);
    if(isPODI){
      bool bFront = bAux.front();
      if(aux==a[i0][next]) bAux.push_front(bFront);
      else bAux.push_front(!bFront);
    }
    i0=next;
  }
}

for(unsigned i=0; i<n; i++){
  p.push_back(pAux.front()); pAux.pop_front();
  if(isPODI){
    b.push_back(bAux.front());
    bAux.pop_front();
  }
}

return cost;


int main(int argc, char ** argv){
ofstream out("report.txt");
if(argc != 7) {
  out << "Numar incorect de argumente pentru program!";
  return 0;
}

unsigned n1, n2, sn, k1, k2, sk;
unsigned n, k;
n1 = atoi(argv[1]);
n2 = atoi(argv[2]);
sn = atoi(argv[3]);
k1 = atoi(argv[4]);
k2 = atoi(argv[5]);
sk = atoi(argv[6]);

short D=5;
out.width(4*D); out << "POD";
out.width(5*D); out << "PODI" << endl;
out.width(D); out<<"n"; out.width(D); out<<"k";
```

```
  out << " |";
  out.width(D); out<<"RAN"; out.width(D); out<<"EX";
  out.width(D); out<<"GMS";
  out.width(D); out<<"GM"; out.width(D); out<<"LB";
  out << " |";
  out.width(D); out<<"RAN"; out.width(D); out<<"EX";
  out.width(D); out<<"GMS";
  out.width(D); out<<"GM"; out.width(D); out<<"LB";
  out << endl;

  for(n=n1; n<=n2; n+=sn)
    for(k=k1; k<=k2; k+=sk){

    vector<BitString> w;
    Permutation p;
    BitString b;
    MMUU H;
    vector<vector<unsigned> > a(n, vector<unsigned>(n, 0));
    generateWords(n, k, w);
    constructMatrix(w, a);

    out.width(D); out << n;
    out.width(D); out << k;
    out << " |";
    out.width(D); out << getRandomCostPOD(n, k, a, p);
    out.width(D); out << getExactPOD(n, k, a, p);
    out.width(D); out << getGMS(n, k, a, false, p, b);
    out.width(D); out << getGM(n, k, a, false, p, b);
    out.width(D); out << getLB(n, k,  a, false, H);
    out << " |";
    out.width(D); out << getRandomCostPODI(n, k, a, p, b);
    out.width(D); out << getExactPODI(n, k, a, p, b);
    out.width(D); out << getGMS(n, k, a, true, p, b);
    out.width(D); out << getGM(n, k, a, true, p, b);
    out.width(D); out << getLB(n, k,  a, true, H);
    out << endl;
  }
  return 0;
}
```

Interpretarea rezultatelor

Tabelul următor conţine rezultatele obţinute de algoritmii *RAN, EX, GMS, GM* şi *LB* pentru instanţe mici ale problemei, folosind programul anterior.

Rezultate POD/PODI *pentru instanțe de dimensiune redusă*

		POD					PODI				
n	k	RAN	EX	GMS	GM	LB	RAN	EX	GMS	GM	LB
5	25	48	31	41	40	37	53	30	38	35	34
5	45	84	75	77	75	75	96	74	77	74	74
5	65	130	116	119	119	113	144	111	112	111	111
6	25	69	50	52	50	50	63	44	48	44	44
6	45	98	91	96	93	91	106	82	84	82	82
6	65	156	118	148	136	134	156	113	137	134	130
7	25	68	58	60	59	56	80	56	59	56	55
7	45	143	116	116	116	115	133	104	114	104	102
7	65	204	180	185	180	179	179	158	166	159	155
8	25	81	73	75	74	69	85	64	70	65	62
8	45	149	141	146	142	140	153	127	135	127	125
8	65	235	204	208	208	202	217	185	195	189	183

Remarcăm că, pentru aceeaşi instanță a problemei, rezultatele algoritmului *EX* pentru *PODI* sunt, aşa cum era de aşteptat, mai bune decât cele pentru *POD* (datorită gradului mai mare de libertate adăugat acesteia). Algoritmul de mărginire inferioară, *LB*, furnizează rezultate mai mici, dar destul de apropiate de cele exacte. Îmbunătățirea adusă de *PODI* în comparație cu *POD* este cu atât mai mare, cu cât lungimea k a cuvintelor este mai mare.

Pentru a compara cei doi algoritmi *Greedy* pe instanțe mari ale problemelor, putem să le apelăm în aceeaşi manieră; de exemplu, pentru n, din mulțimea {1250, 1450, ..., 2050}, iar pentru fiecare astfel de n, valorile lui k din mulțimea {15, 315, 615}. Algoritmii aplicați fiecărei probleme: *Random* (*RAN*), *Greedy Min* (*GM*), *Greedy Min Simplified* (*GMS*) şi *mărginire inferioară* (*LB*). Vom introduce delta (Δ), ce reprezintă deviația lui *GMS* în raport cu *GM*. Pentru un studiu mai exact al comportării celor doi algoritmi, pentru fiecare pereche (n, k) au fost aplicate 15 rulări şi valorile medii sunt completate în tabele. Datorită restricției de a opera numai la un capăt al secvenței curente, în cadrul algoritmului *GMS*, acesta este mai rapid decât *GM* (aproape de două ori, când se consideră matricea de adiacență construită).

Rezultate POD

Input Data		POD				
n	k	RAN	GM	GMS	Δ	LB
1250	15	9341	2480	**2478**	-2	2102
1250	315	196999	**163523**	163535	+12	160951
1250	615	383842	337871	**337861**	-10	334211
1450	15	10868	2766	**2755**	-11	2342
1450	315	228241	189220	**189217**	-3	186318
1450	615	445842	**391117**	391146	+29	386907
1650	15	12363	**3031**	3041	+10	2582
1650	315	259686	**214691**	214698	+7	211377
1650	615	507187	444244	**444233**	-11	439538
1850	15	13915	3315	**3314**	-1	2890
1850	315	291342	**239950**	239984	+34	236325
1850	615	568827	**497320**	497371	+51	492047
2050	15	15277	**3570**	3588	+18	3025
2050	315	322854	**265499**	265533	+34	261566
2050	615	630214	**550324**	550376	+52	544575

Remarcăm că algoritmii *Greedy* furnizează rezultate foarte bune comparativ cu cei *Random* şi că rezultatele produse de algoritmul *LB* sunt, de asemenea, destul de apropiate. Sunt 6 cazuri din 15 în care *GMS* furnizează, în medie, rezultate mai bune decât *GM* şi, pentru celelalte, diferenţa Δ este neglijabilă, raportat la lungimea cuvintelor.

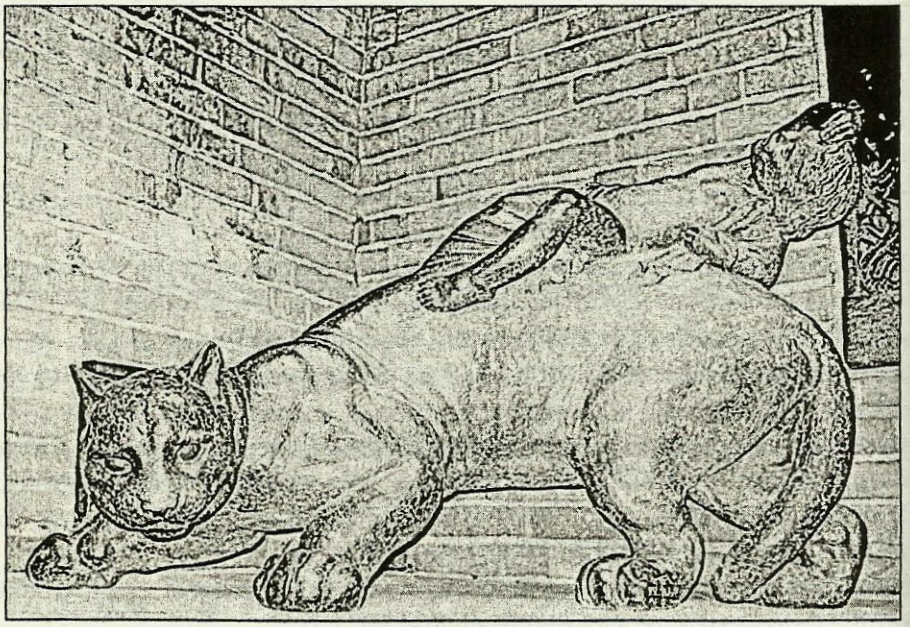
Rezultate PODI

Input Data		PODI				
n	k	RAN	GM	GMS	Δ	LB
1250	15	9372	**2083**	2087	+4	1709
1250	315	196813	**161427**	161481	+44	159040
1250	615	384281	**334411**	334452	+41	330858
1450	15	54189	**2308**	2315	+7	1803
1450	315	228251	**186626**	186702	+76	183975
1450	615	445764	387119	**387106**	-13	383004
1650	15	12295	2522	**2513**	-9	1889
1650	315	259779	211935	**211933**	-2	208841
1650	615	506744	**439712**	439734	+22	435103
1850	15	13836	2750	**2741**	-9	1969
1850	315	291321	**237099**	237102	+3	233542
1850	615	568973	**492231**	492307	+76	487102
2050	15	15366	2952	**2950**	-2	1975
2050	315	322770	**262273**	262282	+9	258412
2050	615	260096	**544788**	544804	+16	539004

Pentru cazul *PODI*, remarcăm că, în 5 cazuri din 15, *GMS* este mai bun decât *GM* şi celelalte rezultate furnizate de *GM* sunt neglijabil mai bune decât cele furnizate de *GMS*. Observăm, de asemenea, că rezultatele furnizate de *RAN* sunt cu atât mai ineficiente, cu cât lungimea k a cuvintelor este mai mică. Ambele *GM* şi *GMS* produc rezultate apropiate de algoritmul de mărginire inferioară, *LB*.

Probleme propuse

1. Care sunt relaţiile ce trebuie să fie satisfăcute de o *metrică*? Demonstraţi că distanţa Hamming este o metrică.
2. Deoarece matricea de adiacenţă este simetrică şi diagonala principală conţine numai zerouri, este suficientă doar reţinerea elementelor din zona superioară a acesteia. Modificaţi programul, astfel încât matricea să fie mapată într-un vector care să conţină doar $(n-1)+(n-2)+\ldots+1 = n(n-1)/2$ elemente.
3. Extindeţi toate metodele cu validarea parametrilor.
4. Formulaţi în preudocod algoritmii *GM*, *GMS* şi *LB*.
5. Extindeţi programul de mai sus, astfel încât să prezinte pentru fiecare caz de test şi raportul detaliat: cuvintele generate, matricea de adiacenţă, permutările, asignările, arborele parţial minim pentru algoritmii rulaţi.
6. Extindeţi programul, astfel încât să se poată executa şi citirea dintr-un fişier a instanţelor problemei.
7. Compactaţi implementările *GM* şi *GMS* prin reţinerea completă a informaţiei despre următorul cuvânt adăugat, fără a mai fi necesar să fie ulterior regăsite.
8. Extindeţi programul şi cu determinarea timpului de execuţie pentru fiecare metodă de rezolvare a *POD/PODI*.
9. Scrieţi un algoritm ce determină succesoarea şi predecesoarea unei permutări în ordine lexicografică, fără a folosi facilităţile *STL*.
   Notă: un astfel de algoritm se găseşte, de exemplu, în [Log01], p. 83.
10. Scrieţi programe de test chiar pentru instanţe mai mari ale problemei.
11. Implementaţi algoritmul *LB* de aflare a arborelui parţial minim, folosindu-l pe cel al lui Prim, în locul celui al lui Kruskal.
12. Citiţi în *Help* metodele şi exemple, pentru tipul *std::list*.
13. Listaţi şi completaţi cu noi exemple toţi algoritmii din *header*-ul *<algorithm>* utilizaţi (*random_shuffle, find, next_permutation*).
14. Metoda *atoi()* transformă un şir de caractere ce reprezintă un număr într[illegible] în numărul întreg corespunzător. Implementaţi o funcţie cu acelaşi efect.



Statuie din Bremen

Capitolul 8
Recursivitatea

12 probleme complet rezolvate, 36 de probleme propuse

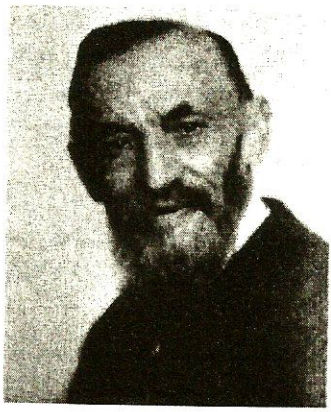
- Inducția matematică
- Recursivitatea. Fundamente
- Suma cifrelor şi oglinditul unui număr natural
- Numărul 4
- *Big Mod*
- Tortul (recursivitate liniară)
- Funcția Ackermann (recursivitate imbricată – *compound recursion*)
- Transformare recursivă în altă bază (din baza 10 în baza p)
- Suma a două rădăcini (recursivitate ramificată)
- Funcția lui Collatz (recursivitate nemonotonă)
- Pătrate şi pătrățele
- Pătrate (recursivitate directă)
- Pătrate şi cercuri (recursivitate indirectă)
- Curba fulgului de zăpadă a lui Koch

Ceea ce este slab este uşor de dezbinat. Ceea ce este mărunt este uşor de risipit. Trebuie să începem să facem ordine când încă nu este dezordine. Un copac mare creşte din unul mic. Un turn cu nouă etaje începe să se construiască dintr-un pumn de pământ. O călătorie lungă începe cu primul pas.

Lao Zi

Inducția matematică

Deseori se cere demonstrarea unor proprietăți nu numai pentru mulțimi finite de numere, ci pentru mulțimi infinite; de exemplu, pentru mulțimea numerelor naturale, $\mathbb{N}$. Pentru a demonstra o propoziție pentru toate numerele naturale, nu este posibilă verificarea ei pentru toate elementele din $\mathbb{N}$, deoarece acest proces nu ar avea sfârşit. Vom descrie în continuare mulțimea numerelor naturale $\mathbb{N}$ folosind axiomele lui Peano, denumite după matematicianul italian Giuseppe Peano (1858-1932), care le-a descoperit.



Giuseppe Peano

P I: 0 este un număr natural.

P II: Fiecare număr natural are exact un succesor, care este tot un număr natural.

P III: 0 nu este succesorul nici unui număr natural.

P IV: Două numere naturale diferite au succesori diferiți.

P V: (axioma inductivă): O mulțime X conține toate numerele naturale atunci când ea conține numărul 0 şi pentru fiecare număr din mulțime ea îl conține şi pe succesorul acestuia.

Axioma inductivă este baza pentru metoda inducției complete. De obicei, această metodă este folosită pentru mulțimi infinite (de exemplu, mulțimea numerelor naturale, mulțimea numerelor pare, cea a numerelor prime etc.).

Paşii unei demonstrații prin inducție completă:

- *Baza inducției – BI (pasul 1).* Mai întâi, se demonstrează propoziția pentru cazul de bază (de exemplu, pentru $n = 0$ sau $n = 1$).
- *Premisa inducției – PI (pasul 2).* Presupunem că propoziția $P(n)$ este adevărată pentru un număr n din $\mathbb{N}$.

- *Concluzia inducției – CI (pasul 3).* Trebuie demonstrat că şi $P(n+1)$ este adevărată, atunci când $P(n)$ este. Cu alte cuvinte, se va demonstra că, dacă propoziția este adevărată pentru un număr natural, atunci ea este adevărată şi pentru succesorul acestuia.

Acum se ştie că $P(0)$ este adevărată (pasul 1) şi concluzia „dacă $P(n)$, atunci şi $P(n+1)$" (pasul 3).

Deducem că acest fapt este suficient ca să afirmăm că propoziția este adevărată pentru toate numerele naturale. Ea a fost demonstrată pentru 0 şi pe baza concluziei rezultă că ea este adevărată pentru 1, apoi pentru 2 ş.a.m.d.

Considerații:

- Concluzia inducției poate să fie adevărată chiar şi atunci când baza inducției nu a fost demonstrată. Este necesar însă ca ambele să fie demonstrate.
- Cazul de bază nu trebuie să fie mereu 0. Există propoziții care încep cu un număr mai mare sau mai mic decât 0. Baza inducției trebuie să fie demonstrată pentru cel mai mic număr posibil. Pot să fie chiar mai multe cazuri de bază.
- n este doar o variabilă, care, bineînțeles, poate să se numească şi altfel. Pentru propozițiile care conțin mai multe variabile, este necesar să se decidă care dintre ele va fi folosită pentru metoda inducției.

Problema 1. Formula sumei

Demonstrați că $\sum_{k=1}^{n} \frac{1}{k(k+1)} = \frac{n}{n+1}$ pentru orice număr natural $n \geq 1$.

Demonstrație directă. Această formulă poate fi demonstrată direct, prin dezvoltarea sumei:

$$\sum_{k=1}^{n} \frac{1}{k(k+1)} = \sum_{k=1}^{n} \left(\frac{1}{k} - \frac{1}{k+1} \right) = \frac{1}{1} - \frac{1}{2} + \frac{1}{2} - \frac{1}{3} + \ldots + \frac{1}{n} - \frac{1}{n+1} = 1 - \frac{1}{n+1} = \frac{n}{n+1}. \square$$

Demonstrație prin inducție completă. Notăm propoziția $\sum_{k=1}^{n} \frac{1}{k(k+1)} = \frac{n}{n+1}$ cu $P(n)$ şi trebuie să demonstrăm că ea este adevărată pentru orice $n \geq 1$.

Baza inducției. $P(1)$ este adevărată: $\frac{1}{1 \cdot 2} = \frac{1}{2}$.

Premisa. $P(n)$: $\sum_{k=1}^{n} \frac{1}{k(k+1)} = \frac{n}{n+1}$ este adevărată.

Concluzia. Trebuie să demonstrăm că şi $P(n+1)$ este adevărată.

Propoziţia $P(n+1)$ este: $\sum_{k=1}^{n+1} \frac{1}{k(k+1)} = \frac{n+1}{n+2}$. Calculăm termenul stâng al egalităţii:

$$\sum_{k=1}^{n+1} \frac{1}{k(k+1)} = \frac{1}{(n+1)(n+2)} + \sum_{k=1}^{n} \frac{1}{k(k+1)} = \frac{1}{(n+1)(n+2)} + \frac{n}{n+1} =$$

$$= \frac{1+2n+n^2}{(n+1)(n+2)} = \frac{(n+1)^2}{(n+1)(n+2)} = \frac{n+1}{n+2}.$$

Conform principiului inducţiei matematice, rezultă că propoziţia $P(n)$ este adevărată pentru orice număr natural $n \geq 1$. ❑

Problema 2. Divizibilitate prin număr prim

Demonstraţi că propoziţia „Dacă p este un număr prim şi n un număr natural, atunci $n^p - n$ se divide cu p" este adevărată.

Demonstraţie prin inducţie matematică. Notăm propoziţia cu $P(n)$.

Baza inducţiei. $P(0)$ şi $P(1)$ sunt adevărate: $n^p - n = 0$ şi 0 divizibil cu p.

Premisa. $P(n)$: „$n^p - n$ divizibil cu p" este adevărată.

Concluzia. Trebuie să demonstrăm că şi $P(n+1)$ este adevărată.

Cu ajutorul binomului lui Newton, scriem $(n+1)^p - (n+1)$ în următoarea formă:

$$(n+1)^p - (n+1) = (n^p + C_p^1 n^{p-1} + C_p^2 n^{p-2} + \ldots + C_p^{p-1} n + 1) - (n+1) =$$

$$= (n^p - n) + C_p^1 n^{p-1} + C_p^2 n^{p-2} + \ldots + C_p^{p-1} n. \qquad (2)$$

Deoarece p este un număr prim, rezultă că toate C_p^1, C_p^2, ..., C_p^{p-1} sunt divizibile cu p.

Din premisa inducției, rezultă că și $n^p - n$ este divizibil cu p. Rezultă că toți termenii din partea dreaptă a formulei (2) sunt divizibili cu p, de unde deducem că și $P(n+1)$ este adevărată.

Conform principiului inducției matematice rezultă că propoziția $P(n)$ este adevărată pentru toate numerele naturale. ❑

Problema 3. Inegalitatea Cauchy-Buniakowski-Schwarz

Prezentăm aici o inegalitate utilă, care este utilizată în multe domenii: algebra liniară (vectori), analiză (serii infinite), teoria probabilității, integrarea produselor. Dacă a_1, $a_2, \ldots, a_n, b_1, b_2, \ldots, b_n$ sunt numere reale, atunci, pentru orice $n \geq 1$, are loc:

$$(a_1^2 + a_2^2 + \ldots + a_n^2)(b_1^2 + b_2^2 + \ldots + b_n^2) \geq (a_1b_1 + a_2b_2 + \ldots + a_nb_n)^2. \quad (3)$$

Demonstrație directă. Pentru aceasta, vom folosi următoarea ecuație de gradul al doilea cu necunoscuta x:

$$(a_1x + b_1)^2 + (a_2x + b_2)^2 + \ldots + (a_nx + b_n)^2 = 0 \quad (4)$$

Ecuația (4) are maxim o rădăcină (partea stângă este mereu mai mare sau egală cu 0 pentru toate numerele reale x) și poate fi scrisă în următoarea formă echivalentă:

$$(a_1^2 + a_2^2 + \ldots + a_n^2)x^2 + 2(a_1b_1 + a_2b_2 + \ldots + a_nb_n)x + (b_1^2 + b_2^2 + \ldots + b_n^2) = 0. \quad (5)$$

Pentru că această ecuație are o singură rădăcină, este necesar ca determinantul ei, Δ, să fie mai mic sau egal cu 0:

$$\Delta = 4(a_1b_1 + a_2b_2 + \ldots + a_nb_n)^2 - 4(a_1^2 + a_2^2 + \ldots + a_n^2)(b_1^2 + b_2^2 + \ldots + b_n^2) \leq 0. \quad (6)$$

Iar (6) este echivalentă cu (3). ❑

Demonstrație prin inducție completă. Notăm (3) cu $P(n)$.

Baza inducției. $P(1)$ este adevărată: $a_1^2b_1^2 \geq (a_1b_1)^2 = a_1^2b_1^2$ pentru orice $a_1, b_1 \in \mathbb{R}$.

Premisa. $P(n)$ este adevărată.

Concluzia. Trebuie să arătăm că și $P(n+1)$ este adevărată.

$P(n+1)$: $(a_1^2+a_2^2+\ldots+a_{n+1}^2)(b_1^2+b_2^2+\ldots+b_{n+1}^2) \geq (a_1b_1+a_2b_2+\ldots+a_{n+1}b_{n+1})^2$

$$\Leftrightarrow \begin{cases} (a_1^2+a_2^2+\ldots+a_n^2)(b_1^2+b_2^2+\ldots+b_n^2)+a_{n+1}^2(b_1^2+b_2^2+\ldots+b_n^2)+ \\ +b_{n+1}^2(a_1^2+a_2^2+\ldots+a_n^2)+a_{n+1}^2b_{n+1}^2 \geq (a_1b_1+a_2b_2+\ldots+a_{n+1}b_{n+1})^2 \end{cases} \quad (7)$$

Pe baza *premisei* rezultă:

$$\sqrt{a_1^2+a_2^2+\ldots+a_n^2}\cdot\sqrt{b_1^2+b_2^2+\ldots+b_n^2} \geq a_1b_1+\ldots+a_nb_n$$
$$\Rightarrow 2a_{n+1}b_{n+1}(a_1b_1+\ldots+a_nb_n) \leq 2a_{n+1}b_{n+1}\sqrt{a_1^2+a_2^2+\ldots+a_n^2}\cdot\sqrt{b_1^2+b_2^2+\ldots+b_n^2} \leq$$
$$a_{n+1}^2(b_1^2+\ldots+b_n^2)+b_{n+1}^2(a_1^2+\ldots+a_n^2) \quad (8)$$

Obținem deci:

$(a_1^2+a_2^2+\ldots+a_n^2)(b_1^2+b_2^2+\ldots+b_n^2) \geq (a_1b_1+a_2b_2+\ldots+a_nb_n)^2$

$a_{n+1}^2(b_1^2+\ldots+b_n^2)+b_{n+1}^2(a_1^2+\ldots+a_n^2) \geq 2a_{n+1}b_{n+1}(a_1b_1+\ldots+a_nb_n)$

$a_{n+1}^2b_{n+1}^2 = a_{n+1}^2b_{n+1}^2$

Atunci când adunăm cele trei relații, obținem:

Ultima inegalitate este adevărată, deoarece

$2xy \leq x^2+y^2$ pentru orice $x, y \in \mathbb{R}$

$(\Leftrightarrow (x-y)^2 \geq 0)$

$(a_1^2+a_2^2+\ldots+a_n^2+a_{n+1}^2)(b_1^2+b_2^2+\ldots+b_n^2+b_{n+1}^2) \geq (a_1b_1+a_2b_2+\ldots+a_nb_n+a_{n+1}b_{n+1})^2$,

și aceasta este $P(n+1)$.

Conform principiului inducției matematice, rezultă că propoziția $P(n)$ este adevărată pentru toate numerele naturale $n \geq 1$. ❑

Probleme propuse

1. Demonstrați următoarele propoziții cu ajutorul inducției matematice și scrieți corespunzător programe care le verifică în general:

a) $\sum_{k=1}^{n} \frac{k}{(2k-1)(2k+1)(2k+3)} = \frac{n(n+1)}{2(2n+1)(2n+3)}$ pentru orice $n \geq 1$.

b) $\sum_{k=1}^{n} \frac{k^4}{(2k-1)(2k+1)} = \frac{n(n+1)(n^2+n+1)}{6(2n+1)}$ pentru orice $n \geq 1$.

2. Găsiți formulele generale pentru următoarele formule și demonstrați-le prin inducție:

a) $S_n = 1\cdot 1! + 2\cdot 2! + \ldots + n\cdot n! = \sum_{k=1}^{n} k\cdot k!$ pentru orice $n \geq 1$.

b) $P_n = \left(1-\frac{1}{4}\right)\cdot\ldots\cdot\left(1-\frac{1}{n^2}\right) = \prod_{k=2}^{n}\left(1-\frac{1}{k^2}\right)$ pentru orice $n \geq 2$.

3. Fie x un număr real, aşa încât $x \neq \pm 1$. Demonstraţi pentru orice $n \in \mathbb{N}$:
$$\frac{1}{1+x}+\frac{2}{1+x^2}+\ldots+\frac{2^n}{1+x^{2^n}} = \frac{1}{x-1}+\frac{2^{n+1}}{1-x^{2^{n+1}}}.$$

4. Demonstraţi că, pentru orice număr natural $n \geq 1$, are loc:

a) $\cos\alpha\cdot\cos 2\alpha\cdot\ldots\cdot\cos 2^n\alpha = \frac{\sin 2^{n+1}\alpha}{2^{n+1}\sin\alpha}$

b) $\sin\alpha+\sin 2\alpha+\ldots+\sin n\alpha = \frac{\sin\frac{n+1}{2}\alpha}{\sin\frac{\alpha}{2}}\sin\frac{n\alpha}{2}$.

5. Inegalitatea lui Bernoulli. Demonstraţi că, pentru orice număr real x, astfel încât $x > -1$ şi $n \in \mathbb{N}$, are loc: $1+nx \leq (1+x)^n$.

6. Demonstraţi că, pentru orice număr natural $n \geq 1$, au loc:

a) $11^{n+2}+12^{2n+1}$ divizibil cu 133;

b) $4^n+15n-1$ divizibil cu 9;

c) $3\cdot 5^{2n+1}+2^{3n+1}$ divizibil cu 17;

d) $2^{7n+3}+3^{2n+1}\cdot 5^{4n+1}$ divizibil cu 23.

7. Şirul lui Fibonacci este definit astfel: $F(0) = 0$, $F(1) = 1$, $F(n) = F(n-1) + F(n-2)$ pentru orice $n \geq 2$. Demonstraţi că:

a) $F(n+m) = F(m+1)\cdot F(n)+F(m)\cdot F(n-1)$ pentru orice $m, n \in \mathbb{N} - \{0\}$.

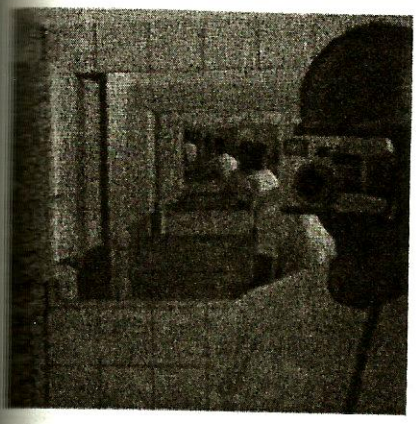
b) $F(n\cdot k)$ este divizibil cu $F(n)$ pentru orice $n, k \in \mathbb{N} - \{0\}$.

c) Pentru orice $n, k \in \mathbb{N} - \{0\}$ au loc: $F(kn-1) \equiv F^k(n-1) \pmod{F^2(n)}$ şi $F(kn-2) \equiv (-1)^{k+1}F^k(n-2) \pmod{F^2(n)}$.

d) $F_{n\cdot F(n)}$ este divizibil cu $F^2(n)$ pentru orice $n \in \mathbb{N} - \{0\}$.

e) $F(n\cdot F^m(n))$ divizibil cu $F^{m+1}(n)$ pentru orice $m \in \mathbb{N}$ şi $n \in \mathbb{N} - \{0\}$.

Recursivitatea. Fundamente



Termenul recursivitate (în latină, *recurrere* = „a reveni", „a se întoarce") înseamnă utilizarea a „ceva" „cu referire la el însuşi". Recursivitatea este pentru informatică ceea ce inducţia completă este în matematică. Numim recursivă o structură atunci când părţi ale sale au acelaşi mod de construcţie ca şi structura însăşi; de exemplu, arborii şi listele sunt structuri de date recursive. Un algoritm este recursiv atunci când în interiorul său există subprobleme care apelează la acelaşi algoritm. Cunoscuta problemă a turnurilor din Hanoi, de exemplu, se formulează recursiv foarte elegant. Algoritmii iterativi se pot transforma în algoritmi recursivi (pentru o buclă se creează o metodă recursivă) şi invers (se foloseşte o stivă ce reprezintă apelurile recursive).

Pentru a scrie o metodă recursivă, este necesar să decidem ce se întâmplă pentru un anumit nivel, deoarece la toate nivelurile se întâmplă acelaşi lucru. Foarte importantă este condiţia de terminare (trebuie să fim siguri că algoritmul se termină!); aceasta înseamnă că avem nevoie de o condiţie care decide că metoda nu va mai fi apelată. Fără această condiţie se ajunge la o buclă infinită.

Există mai multe tipuri de algoritmi recursivi, care pot fi, de exemplu, clasificaţi după numărul aşteptat de apeluri recursive în raport cu dimensiunea variabilelor de control:

- *recursivitate liniară*: fiecare apel al funcţiei recursive se apelează pe el însuşi cel mult o singură dată, numărul apelurilor recursive depinde liniar de dimensiunea variabilelor de control. Exemple: calculul factorialului, suma cifrelor, transformarea unui număr în alt sistem de numeraţie;
- *recursivitate ramificată*: fiecare apel al funcţiei recursive conţine cel puţin două apeluri ale aceleiaşi funcţii. În acest caz, trebuie să fim atenţi ca aceeaşi valoare să nu fie calculată de mai multe ori, situaţie ce ar conduce la timpi de execuţie extrem de mari. Exemple: numerele Fibonacci, coeficienţii binomiali;
- *recursivitate imbricată* (*compound recursion*): argumentul pentru apelul recursiv apelează însuşi din nou metoda recursivă. Astfel, apar foarte multe apeluri ale funcţiei. Exemple: funcţiile Ackermann, Manna-Pnueli;

- *recursivitate deschisă (nemonotonă)*: argumentul de control nu se dezvoltă mereu în direcția condiției de terminare. Exemplu: șirul ($3n + 1$) este cunoscut ca și funcția Collatz.

După tipul apelului, algoritmii recursivi pot fi astfel clasificați:
- *recursivitate directă*: o metodă se apelează direct pe ea însăși.
- *recursivitate indirectă*: cel puțin două metode care se apelează reciproc.

Problema 1. Suma cifrelor și oglinditul unui număr natural

Scrieți metode recursive pentru determinarea sumei cifrelor și a oglinditului unui număr natural dat n ($n > 0$). Oglinditul unui număr natural se obține prin inversarea cifrelor sale. Exemplu:

Tastatură	Ecran
n= 34690213776	sumDigits(n)= 48 reverse(n)= 67731209643

Analiza problemei și proiectarea soluției

Pentru rezolvarea problemei, vom scrie metodele recursive *sumDigits()*, *noDigits()*, *pow()* și *reverse()*. Citiți explicațiile din casetele alăturate. Metoda de oglindire a unui număr natural funcționează astfel:
- Se adaugă ultimei cifre numărul necesar de zerouri.
- Acest număr se adaugă oglinditul numărului obținut prin eliminarea ultimei cifre:

*((n mod 10)*pow(10,noDigits(n) – 1)+reverse(n div 10)).*

Program

```
#include <iostream>

short sumDigits(unsigned long long n){
  if(n)
    return (n%10+sumDigits(n/10));
  else
    return 0;
}

short noDigits(unsigned long long n){
  if(n)
    return 1+noDigits(n/10);
  else
    return 0;
}
```

Suma cifrelor unui număr natural este suma ultimei cifre (*n mod* 10) și suma cifrelor numărului obținut prin eliminarea acestei ultime cifre (*n div* 10).

Numărul cifrelor unui număr natural este 1 plus numărul cifrelor numărului obținut prin eliminarea ultimei cifre (*n div* 10).

```
unsigned long long pow(short b, short exp){
  if(exp)
    return b*pow(b, exp-1);
  else
    return 1;
}

unsigned long long reverse(unsigned long long n){
  if(n)
    return ((n%10)*pow(10,noDigits(n)-1)+reverse(n/10));
  else
    return 0;
}

int main(){
  unsigned long long n;
  std::cout << "n= "; std::cin >> n;
  std::cout << "sumDigits(n)= " << sumDigits(n) << std::endl;
  std::cout << "reverse(n)= " << reverse(n);
  return 0;
}
```

$$b^n = \begin{cases} 1, \text{ pentru } n = 0 \\ b \cdot b^{n-1}, \text{ pentru } n > 0 \end{cases}$$

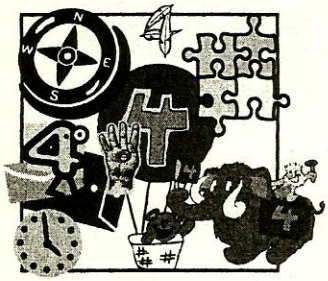
Probleme propuse

1. Scrieți pe hârtie valorile pentru pașii executați de program relativ la exemplul dat.
2. Implementați metode iterative pentru metodele recursive *sumDigits()*, *noDigits()*, *pow()* și *reverse()*.
3. Implementați metoda de ridicare rapidă la putere, pe baza algoritmului următor:

> ***Ridicarea rapidă la putere*** ($O(\log n)$, unde n este exponentul)
>
> 1. $e = e_0 + e_1 \cdot 2 + e_2 \cdot 2^2 + \ldots + e_r \cdot 2^r$
>
> cu $e_0, e_1, \ldots, e_r \in \{0, 1\}$ (reprezentarea binară a lui e)
>
> 2. $e = e_0 + 2 \cdot (e_1 + 2 \cdot (e_2 + 2 \cdot (\ldots + 2(e_{r-1} + 2 \cdot e_r))\ldots))$
> 3. $b^e = (\ldots((((b^{e_r})^2 \cdot b^{e_{r-1}})^2 \cdot b^{e_{r-2}})^2 \cdot b^{e_{r-3}}) \cdot \ldots)^2 \cdot b^{e_0}$
> 4. La fiecare pas, rezultatul parțial este ridicat la putere și apoi multiplicat cu 1 sau cu b, după cum e_i este 0, respectiv 1.

4. Scrieți metode iterative și recursive pentru calculul valorilor $n! = 1 \cdot 2 \cdot \ldots \cdot n$ și $S(n) = 1 + 2 + \ldots + n$.
5. Scrieți o metodă iterativă și una recursivă pentru calculul sumei $S(m, n) = m + (m + 1) + \ldots + n$ (dacă $m > n$, atunci returnați 0).

Problema 2. Numărul 4



Numărul patru simbolizează cele patru puncte cardinale, cele patru anotimpuri şi cele patru elemente.

Se poate demonstra că din numărul patru se poate obține orice număr natural cu ajutorul următoarelor operații:

A) se adaugă un 4 la sfârşit;

B) se adaugă un 0 la sfârşit;

C) numărul curent se împarte la 2.

De exemplu, numărul 2354 se obține din numărul 4 aplicând succesiv operațiile *CCBCBACA*. Scrieți un program care să arate pentru numerele naturale din fişierul *nr4.in* cum se pot obține acestea succesiv din numărul 4, folosind operațiile A-C. Rezultatele se vor scrie în fişierul *nr4.out*. Exemplu:

nr4.in	nr4.out
2524	4->2->1->10->5->50->504->252->2524
564	4->44->22->224->112->56->564
12	4->2->24->12
3	4->2->24->12->6->3

(Inspirată din revista sovietică Kvant, *autor A.K. Tolpigo)*

Analiza problemei şi proiectarea soluției

Vom determina drumul către numărul n prin construirea drumului invers de la n la numărul 4. Operațiile inverse sunt:

A') se elimină cifra 4 de la sfârşit, dacă este un 4;

B') se elimină cifra 0 de la sfârşit, dacă este 0;

C') se înmulțeşte cu 2.

Demonstrați că din orice număr natural n se poate obține numărul 4, cu ajutorul operațiilor A-C.

Putem să clasificăm numărul n după ultima cifră şi să demonstrăm pentru fiecare categorie. Este însă suficient dacă demonstrăm că din orice număr par $n \geq 4$ se poate ajunge la un număr par mai mic decât el. Dacă n este un număr impar, atunci el se poate tranforma în număr par prin înmulțirea cu 2. Dupa ultima cifră a numărului par n, sunt posibile următoarele transformări:

i) $10k \to k$;

ii) $10k + 2 \to 20k + 4 \to 2k$;

iii) $10k + 4 \to k \to 2k$;

iv) $10k + 6 \to 20k + 10 + 2 \to 40k + 20 + 4 \to 4k + 2$;

v) $10k + 8 \to 20k + 10 + 6 \to 40k + 30 + 2 \to 80k + 60 + 4 \to 8k + 6$. ❑

Vom implementa metoda recursivă *numberFour()*.

Program

```
#include <fstream>

void numberFour(int n, std::ofstream &out){
  if(n-4){
    switch(n%10){
      case 0:  numberFour(n/10, out); break;
      case 4:  numberFour(n/10, out); break;
      default: numberFour(n*2, out);
    }
    out << "->" << n;
  }


int main(){
  int n;
  std::ifstream in("nr4.in");
  std::ofstream out("nr4.out");
  while(in && !in.eof() && in>>n){
    out << "\n4";
    numberFour(n, out);
  }
  return 0;
}
```

Probleme propuse

1. Notați pe hârtie pașii făcuți de algoritm pentru numerele din exemplu.
2. Scrieți o metodă iterativă pentru rezolvarea problemei.

Problema 3. *Big Mod*

Calculați valoarea $R = B^P \bmod M$ pentru B, P și M numere naturale, așa încât $0 \leq B, P \leq$ 000.000 și $0 \leq M \leq 50.000$.

Date de intrare. În fișierul *bigmod.in* se găsesc mai multe instanțe ale problemei ca triplete, câte unul pe linie.

Date de ieșire. Scrieți în fișierul de ieșire *bigmod.out* pentru fiecare triplet (B, P, M) valoarea $R = B^P \bmod M$. Exemplu:

bigmod.in	bigmod.out
18132 17	13
1765 3	2
4859 3029382 36123	13195

(http://acm.uva.es/p/v3/374.html)

Analiza problemei şi proiectarea soluției

Propoziție. Pentru orice numere $a, b, c \in \mathbb{N} - \{0\}$, are loc $(a \cdot b) \bmod c = ((a \bmod c) \cdot (b \bmod c)) \bmod c$.

Demonstrație. Conform teoremei împărțirii cu rest, obținem succesiv:

$$\exists R \in \{0, ..., c-1\}, \text{a.î. } a \cdot b = Q \cdot c + R \rightarrow \quad (1)$$
$$R = (a \cdot b)\%c = a \cdot b - Q \cdot c$$

$$\exists R_1 \in \{0, ..., c-1\}, \text{a.î. } a = Q_1 \cdot c + R_1, R_1 = a\%c$$
$$\exists R_2 \in \{0, ..., c-1\}, \text{a.î. } b = Q_2 \cdot c + R_2, R_2 = b\%c \quad \Rightarrow a \cdot b = Q_3 \cdot c + R_1 \cdot R_2 \Rightarrow$$

$$\Rightarrow (a \cdot b)\%c = (R_1 \cdot R_2)\%c = (a\%c + b\%c)\%c \,. \square$$

Vom scrie metoda recursivă *bigMod()*, care se bazează pe această propoziție:

$$bigMod(B, P, M) = \begin{cases} \text{a) } 1, \text{ dacă } P = 0 \\ \text{b) } 0, \text{ dacă } a \text{ neadevărată şi } B = 0 \\ \text{c) } B\%M, \text{ pentru } a \text{ şi } b \text{ neadevărate şi } (B = 0 \text{ sau } P = 1) \\ \text{d) } (bigMod(B, P/2, M)^2)\%M, \text{ pentru } a, b \text{ şi } c \text{ neadevărate şi } P \text{ număr par} \\ \text{e) } (bigMod(B, P-1, M) \cdot (B\%M))\%M, \text{ pentru } a, b, c \text{ neadevărate şi } P \text{ număr impar} \end{cases}$$

Program

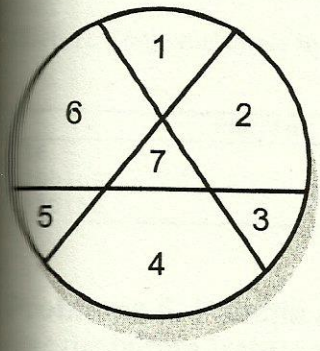
```
#include <fstream>

using namespace std;

unsigned long int bigMod(long int B, long int P, long int M){
  if(0==P) return 1;
  if(0==B) return 0;
  if(1==B || 1==P) return B%M;
  if(1==M) return 0;
  if(0==P%2){
    unsigned long int aux;
    aux = bigMod(B, P/2, M);
    return (aux*aux)%M;
  } else
    return (bigMod(B, P-1, M) * (B%M))%M;
}
```

```
int main(){
  unsigned long int B, P, M;
  ifstream in("bigmod.in");
  ofstream out("bigmod.out");
  while(in && !in.eof() && in>>B>>P>>M)
    out << bigMod(B, P, M) << endl;
  return 0;
```

Probleme propuse

1. Scrieți și o variantă iterativă pentru rezolvarea problemei.
2. Demonstrați propoziția de mai sus astfel: $n \bmod c = n - c(n \text{ div } c)$, $((a \bmod c) \times (b \bmod c)) \bmod c = ((a - c(a \text{ div } c)) \times (b - c(b \text{ div } c))) = \ldots$
3. De la tastatură se citesc n numere naturale $a_1, a_2, \ldots, a_n$, fiecare mai mic decât 2000000, și un număr natural $0 < c < 50000$. Scrieți un program care calculează valoarea $(a_1 \cdot a_2 \cdot \ldots \cdot a_n) \bmod c$.

Problema 4. Tortul (recursivitate liniară)



La o petrecere, trebuie tăiate bucăți de tort care pot avea dimensiuni diferite. Din trei tăieturi, se pot obține șapte bucăți, ca în figura alăturată. Câte bucăți se pot obține din n tăieturi? (Câte regiuni se obțin maximal în plan atunci când se trasează n drepte?)

Date de intrare. În fișierul *tort.in* se găsesc, câte una pe linie, numărul de tăieturi n ($0 \le n \le 2000$).

Date de ieșire. Scrieți în fișierul *tort.out*, pentru fiecare număr de tăieturi din fișierul de intrare, numărul maxim de bucăți de tort care se pot obține cu numărul dat de tăieturi, ca în exemplul de mai jos:

tort.in	tort.out
0 1 2 9 10 2000 678	0 taieturi -> 1 bucati! 1 taieturi -> 2 bucati! 2 taieturi -> 4 bucati! 9 taieturi -> 46 bucati! 10 taieturi -> 56 bucati! 2000 taieturi -> 2001001 bucati! 678 taieturi -> 230182 bucati!

Analiza problemei şi proiectarea soluției

Dacă numărul maxim de regiuni pentru $n-1$ drepte care se intersectează este $S(n-1)$, atunci pentru încă o dreaptă se vor obține în plus încă n regiuni:

$S(0) = 1$ (nici o tăietură → rămâne o bucată: întregul tort)
$S(n) = n + S(n-1)$, pentru orice $n > 0$. (1)

Formula (1) este o formulă recursivă şi vom implementa corespunzător ei metoda $S()$ în program.

Program 4.1.

```
#include <fstream>

int S(int n){
  if(n<1)return 1;
  else
    return n + S(n-1);
}

int main(){
  int n;
  std::ifstream in("tort.in");
  std::ofstream out("tort.out");
  while(in && !in.eof() && in>>n){
    out.width(4);
    out << n << " taieturi -> ";
    out.width(7);
    out << S(n) << " bucati!" << std::endl;
  }
  return 0;
}
```

Observăm că, pentru valori mai mari ale lui n, timpul de execuție al programului creşte foarte mult, deoarece metoda se apelează foarte des. Din formula (1) deducem:

$$S(n) = n + S(n-1) = n + (n-1) + S(n-2) = \ldots = 1 + \frac{n \cdot (n+1)}{2}. \quad (1)$$

Program 4.2.

```
#include <fstream>

int main(){
  long n;
  std::ifstream in("tort.in");
  std::ofstream out("tort.out");
  while(in && !in.eof() && in>>n && n >=0){
    out.width(8);
    out << n << " taieturi -> ";
    out.width(10);
    out << 1 + n*(n+1)/2;
        << " bucati!" << std::endl;
  }
  return 0;
}
```

Problemă propusă

Problema inversă. Se dă numărul maxim de regiuni care se obțin și se cere numărul de tăieturi. Toate numerele în fișierul *reg.in* sunt, așadar, de forma $1+\frac{n(n+1)}{2}$. Exemplu:

reg.in	taieturi.out
56 2001001 230182	10 2000 678

Problema 5. Funcția Ackermann (recursivitate imbricată – *compound recursion*)

Există mai multe funcții matematice care poartă denumirea de Ackermann. Inițial, o astfel de funcție ce crește extrem de repede a fost introdusă de către Wilhelm Ackermann în 1926. Funcția Ackermann este folositoare de exemplu pentru determinarea limitelor în informatica teoretică. Cu ajutorul ei, se pot construi module de test pentru apeluri recursive.

Funcția lui Ackermann este astfel definită pentru $m, n \in \mathbb{N}$:

$$Ack(n,m) = \begin{cases} m+1, \text{dacă } n=0 \\ Ack(n-1,\ 1), \text{dacă } m=0 \\ Ack(n-1,\ Ack(n,\ m-1)), \text{dacă } m \neq 0 \text{ și } n \neq 0 \end{cases}$$

Funcția lui Ackermann crește foarte repede: *Ack*(3, 4) = 125, dar *Ack*(4, 2) are 19729 cifre! Scrieți un program care să calculeze valoarea funcției lui Ackermann pentru perechi (n, m) de numere naturale, știind că această valoare se încadrează în tipul *unsigned long long*. Exemplu:

ack.in	ack.out
0 2 2 0 3 4 3 5 3 2	Ack(0, 2)= 3 Ack(2, 0)= 3 Ack(3, 4)= 125 Ack(3, 5)= 253 Ack(3, 2)= 29

Analiza problemei şi proiectarea soluției

Vom implementa definiția matematică recursivă a funcției cu ajutorul metodei *ack*().

Program

```
#include <fstream>

unsigned long long ack(short n, short m){
  if(0==n) return m+1;
  else if(0==m) return ack(n-1, 1);
  else return ack(n - 1, ack(n, m - 1));
}

int main(){
  short m, n;
  std::ifstream in("ack.in");
  std::ofstream out("ack.out");
  while(in && !in.eof() && in>>n>>m){
    out << "Ack(" << n << ", " << m
        << ")= "  << ack(n, m) << std::endl;
  }
  return 0;
}
```

Problemă propusă

Manna-Pnueli. Scrieți un program care pentru un număr întreg x, astfel încât $-1000 \leq x \leq 1000$, calculează valoarea funcției Manna-Pnueli $f(x)$, definită astfel:

$$f:\mathbb{Z} \to \mathbb{Z}, f(x)=\begin{cases} x-1, \text{dacă } x \geq 12 \\ f(f(x+2)), \text{dacă } x<12 \end{cases}$$

Exemplu:

mp.in	mp.out
12	Manna-Pnueli(12)= 11
34	Manna-Pnueli(34)= 33
56	Manna-Pnueli(56)= 55
-123	Manna-Pnueli(-123)= 11
98	Manna-Pnueli(98)= 97
678	Manna-Pnueli(678)= 677
-234	Manna-Pnueli(-234)= 11

Problema 6. Transformare recursivă în altă bază (din baza 10 în baza *p*)

Scrieți o metodă recursivă care să transforme un număr natural n ($0 \le n \le 9.000.000$) din baza 10 într-o bază dată p ($q \le p \le 9$) Exemplu:

baseP.in	baseP.out
324123534 2	324123534 in base 2 = 10011010100011011101110001110
324539 9	324539 in base 9 = 544158
654789045 6	654789045 in base 6 = 144550222433
675432 4	675432 in base 4 = 2210321220
9000000 8	9000000 in base 8 = 42252100
3 2	3 in base 2 = 11
8 6	8 in base 6 = 12

Analiza problemei şi proiectarea soluției

Dacă n are reprezentarea $a_k a_{k-1}...a_1 a_0$ în baza p, atunci are loc:

$$n = \overline{a_k a_{k-1}...a_1 a_0} = a_k p^k + a_{k-1} p^{k-1} + ... + a_1 p + a_0, \quad a_k, a_{k-1}, ..., a_0 \in \{0, 1, ..., p-1\}$$

Deci $n = p(a_k p^{k-1} + a_{k-1} p^{k-2} + ... + a_1) + a_0$.

De aici, se deduce următorul algoritm recursiv:

```
doTransformBase10ToP(n, p)
  If (p > 0) Then
    doTransformBase10ToP(n div p, p)
    Write (n % p)
  End_If
End_doTransformBase10ToP(n, p)
```

Program

```
#include <fstream>

using namespace std;

void transfBase10ToP(unsigned long long n,
                     short p, ofstream &out){
  if(n){
    transfBase10ToP(n/p, p, out);
    out << n%p;
  }
}

int main(){
  unsigned long long n;
  short p;
  ifstream  in("baseP.in");
  ofstream out("baseP.out");
  while(in && !in.eof() && in>>n>>p){
    out.width(10);
    out << n << " in base " << p
        << " = ";
    transfBase10ToP(n, p, out);
    out << endl;
  }
  return 0;
}
```

Probleme propuse

1. Scrieți un algoritm nerecursiv care să rezolve problema.
2. Implementați un program (variantă recursivă și iterativă) pentru transformarea inversă. Se dă o pereche de numere (n_p, p) și se cere transformarea numărului n_p din baza p în baza 10. Exemplu:

baseP10.in	baseP10.out
42252100 8	9000000
11 2	3
12 6	8

Analizați următoarea implementare:

```
unsigned long long getBasePTo10(string np, short p){
   if(np.length()){
     short aux = np[np.length()-1]-'0';
```

```
    return aux +
     p*getBasePTo10(np.substr(0, np.length()-1), p);
  }else
    return 0;
}
```

Valorile ASCII '0', '1', ..., '9' sunt amplasate succesiv şi de aceea valoarea (*np[np.length() – 1] – '0'*) este de fapt numărul corespunzător 0, 1, 2, ..., 9.

3. Scrieți cel puțin două variante pentru transformarea unui număr natural în binar prin folosirea operatorilor pe biți. Analizați varianta:

```
cin >> n; cout << n << " in base 2 = ";
short SIZ_INT = sizeof(unsigned long long)*8;
for(short i=SIZ_INT-1; i>=0; i--)
   cout << (n>>i&1);                    //bitul al i-lea din n
```

Problema 7. Suma a două rădăcini (recursivitate ramificată)

Se consideră ecuația $x^2 - Sx + P = 0$, cu $S, P \in \mathbb{R}$. Fie x_1 şi x_2 rădăcinile acestei ecuații şi trebuie calculată valoarea $T_n = x_1^n + x_2^n$ pentru $n \in \mathbb{N}$. T_n va trebui calculat fără a folosi valorile explicite ale rădăcinilor x_1 şi x_2. Numerele reale S, P, cu $-500 \le S, P \le 500$, şi numărul natural n ($0 \le n \le 20$) se află în fişierul de intrare, iar T_n trebuie scris în fişierul de ieşire, câte unul pe linie pentru fiecare set de date de intrare. Exemplu:

equation.in	equation.out
5 6.54 3.22 0 3 4 1 3.45 6.78	7799.79 2 3.45

Analiza problemei şi proiectarea soluției

Deoarece x_1 şi x_2 sunt rădăcinile ecuației, rezultă că $x^2 - Sx + P = (x - x_1)(x - x_2)$. Deci $x_1 + x_2 = S$ şi $x_1 \cdot x_2 = P$ (relațiile lui Viète). Aşadar, formula pentru T_n este:

$$T_n = \begin{cases} 2, \text{dacă } n = 0 \\ x_1 + x_2, \text{dacă } n = 1 \\ ST_{n-1} - PT_{n-2}, \text{dacă } n > 1 \end{cases} \quad (1)$$

Program

```
#include <fstream>

double sum(short n, double S, double P){
  if(0==n) return 2;
  if(1==n) return S;
  return
    S*sum(n-1, S, P)-P*sum(n-2, S, P);
}

int main(){
  short n;
  double S, P;
  std::ifstream in("equation.in");
  std::ofstream out("equation.out");
  while(in && !in.eof() && in>>n>>S>>P){
    out.precision(6);
    out << sum(n, S, P) << std::endl;
  }
  return 0;
}
```

Probleme propuse

1. Demonstrați formula (1) prin inducție.
2. *Generalizare.* Se consideră ecuația de gradul k: $x^k - S_1 x^{k-1} + ... + (-1)^k S_k = 0$ cu rădăcinile $x_1, x_2, ..., x_k$ și se cere valoarea $T_n = x_1^n + x_2^n + ... + x_k^n$.
   Indicație. Formula recursivă este $T_n = S_1 T_{n-1} - S_2 T_{n-2} + ... + (-1)^{k-1} \cdot S_k \cdot T_{n-k}$.

Problema 8. Funcția lui Collatz (recursivitate nemonotonă)

Funcția lui Collatz este definită astfel:

$$f(n) = \begin{cases} 1, \text{ pentru } n = 1 \\ \dfrac{n}{2}, \text{ dacă } n \text{ este par} \\ 3 \cdot n + 1, \text{ dacă } n \text{ este impar} \end{cases}$$

și are proprietatea că este „convergentă" către 1. De exemplu, pentru $n = 12$, secvența generată este: 12, 6, 3, 10, 5, 16, 8, 4, 2, 1 și are lungimea 9, deoarece se ajunge în 9 pași la valoarea 1. Scrieți o funcție recursivă, care generează secvența corespunzătoare și afișează și numărul de pași în care s-a ajuns la 1. Exemplu:

numbers.in	collatzSeq.out
1 12 67 1003 234	1 [1] STOP<0> 12 6 3 10 5 16 8 4 2 1 [1] STOP<9> 67 202 101 304 152 76 38 19 58 29 88 44 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1 [1] STOP<27> 1003 3010 1505 4516 2258 1129 3388 1694 847 2542 1271 3814 1907 5722 2861 8584 4292 2146 1073 3220 1610 805 2416 1208 604 302 151 454 227 682 341 1024 512 256 128 64 32 16 8 4 2 1 [1] STOP<41> 234 117 352 176 88 44 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1 [1] STOP<21>

Analiza problemei şi proiectarea soluției

Vom implementa metoda recursivă *fCollatz()*, care, cu ajutorul parametrului *l*, numără și numărul de apeluri ale funcției.

Program

```
#include <fstream>

using namespace std;

void fCollatz(unsigned long n, int& l, ofstream &out){
  out << n << " ";
  if(n!=1){
    l++;
    if(n%2){
      fCollatz(3*n+1, l, out);
      }else{
      fCollatz(n/2, l, out);
    }
    } else{
      out << "[1] STOP";
  }


int main(){
  unsigned long n;
  int l;
  ifstream in("numbers.in");
  ofstream out("collatzSeq.out");
  while(in && !in.eof() && in>>n){
    l=0;
```

```
        fCollatz(n, 1, out);
        out << "<" << 1 << ">" << endl << endl;
    }
    return 0;
}
```

Probleme propuse

1. Scrieți și o variantă recursivă.
2. Notăm cu A operația de „înjumătățire" și cu B operația „$3n + 1$". Dezvoltați programul astfel încât să calculeze și numărul de operații de fiecare tip. De exemplu, pentru $n = 12$ este nevoie de șapte operații A și de două operații B; pentru $n = 1003$ este nevoie de 29 operații A și de 12 operații B. Implementați un program care să prelucreze toate numerele dintr-un interval dat și să le afișeze pe cele care au nevoie de un număr maximal de pași pentru a ajunge la 1 (de exemplu, în intervalul [45, 459] se găsește numărul 327, care are nevoie de 143 de pași). Furnizați de asemenea acele numere din intervalul dat care au nevoie de un număr maxim de operații A, respectiv B.

Problema 9. Pătrate și pătrățele

Un pătrat cu laturile paralele cu axele este unic, dacă se dau centrul său și lungimea laturii. Spunem că un pătrat cu lungimea laturii $2k + 1$ are dimensiunea k. Pentru un număr natural k, definim familia pătratelor cu următoarele proprietăți:

i) cel mai mare pătrat are dimensiunea k (lungimea laturii sale este $2k + 1$) și se află amplasat în centrul unei rețele de dimensiune 1024 (această rețea are lungimea laturii 2049);

ii) pentru dimensiunea k, are loc: $1 \leq k \leq 512$;

iii) toate pătratele care au o dimensiune mai mare decât 1 generează câte un pătrat de dimensiune k div 2 în toate cele patru vârfuri ale sale;

iv) colțul stânga-sus al ecranului are coordonatele (0, 0) și colțul dreapta-jos are coordonatele (2048, 2048).

Pentru un număr natural k, putem să definim în mod unic familia pătratelor cu aceste proprietăți. Un punct pe ecran se poate găsi în nici unul, unul sau mai multe pătrate. Scrieți un program care, pentru un număr natural k dat și coordonatele unui punct în plan, să decidă în câte pătrate din familia k se află acest punct.

Date de intrare. Fiecare linie în fișierul de intrare *patrate.in* conține trei valori k, x_0 și y_0 și reprezintă un set de date de intrare.

Date de ieșire. Scrieți pentru fiecare caz din fișierul de intrare câte o linie în fișierul de ieșire *patrate.out* cu numărul corespunzător de pătrate în care se găsește punctul (x_0, y_0). Exemplu:

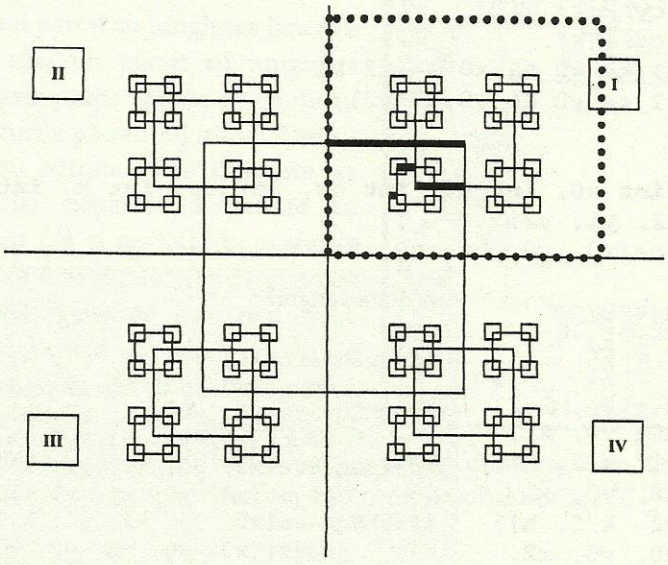
patrate.in	patrate.out
56 1012 1000 500 1000 1000 500 113 941 1100 512 512	1 4 5 5

(ACM, North-Western European Regionals, 1992)

Analiza problemei şi proiectarea soluţiei

O primă metodă de rezolvare ar fi să verificăm recursiv pentru fiecare pătrat dacă el conţine punctul dat, caz în care variabila n se va incrementa. Pentru aceasta, vom scrie metoda *inSquare*(x_0, y_0, cx, cy, k), unde primii doi parametri definesc punctul (x_0, y_0) şi următorii definesc pătratul cu centrul (cx, cy) şi dimensiunea k. Meotda *count()* numără în câte pătrate se află punctul dat şi este apelată recursiv pentru cele patru pătrate care se creează în cele patru vârfuri ale pătratului dat.

Propoziţie. Dacă desenăm şi axele de simetrie prin centrul primului pătrat, atunci toate pătratele generate de vârful dreapta-sus se află în primul cadran. La fel, toate pătratele generate de vârful stânga-sus se află în cel de-al doilea cadran, cele generate de vârful stânga-jos în al treilea, cele generate de vârful dreapta-jos se află în cel de-al patrulea cadran.



Demonstraţie. În cazul în care k este lungimea laturii pătratului punctat, atunci segmentele îngroşate care se apropie de ordonată au lungimile $\frac{k}{2}, \frac{k}{4}, \frac{k}{8}, \ldots, \frac{k}{2^n}, \ldots$. Acestea reprezintă o progresie geometrică; rezultă deci:

$$\frac{k}{2}+\frac{k}{2^2}+\frac{k}{2^3}+...+\frac{k}{2^n}=k\cdot\left(1-\frac{1}{2^n}\right).$$

Şirul $k\cdot\left(1-\frac{1}{2^n}\right)$, pentru $n = 0, 1, 2, ...$, este crescător şi are limita k, de unde rezultă că toate pătratele generate de vârful dreapta-sus se află în primul cadran. Datorită simetriei, deducem de asemenea că şi toate celelalte grupe de pătrate se află în cadranele II, III şi IV. ❑

A două alternativă pentru program apelează metoda recursivă doar pentru pătratul în care se află punctul dat.

Program

```
#include <fstream>

int inSquare(int x0, int y0, int cx, int cy, int k){
  int x1 = cx - k;
  int x2 = cx + k;
  int y1 = cy - k;
  int y2 = cy + k;
  return (x1 <= x0 && x0 <= x2 &&
          y1 <= y0 && y0 <= y2);
}

void count(int x0, int y0, int cx, int cy, int k, int& n){
  int x1, x2, y1, y2;
  if(inSquare(x0, y0, cx, cy, k)) n++;
  if(k>1){
    x1 = cx - k;
    x2 = cx + k;
    y1 = cy - k;
    y2 = cy + k;
    count(x0, y0, x1,
          y1, k/2, n);
    count(x0, y0, x1,
          y2, k/2, n);
    count(x0, y0, x2,
          y1, k/2, n);
    count(x0, y0, x2,
          y2, k/2, n);
  }
}
```

A doua variantă:

```
...
if(inSquare(x0, y0, x1, y1, k))
      count(x0, y0, x1, y1, k/2, n);
if(inSquare(x0, y0, x1, y2, k))
      count(x0, y0, x1, y2, k/2, n);
if(inSquare(x0, y0, x2, y1, k))
      count(x0, y0, x2, y1, k/2, n);
if(inSquare(x0, y0, x2, y2, k))
      count(x0, y0, x2, y2, k/2, n);
...
```

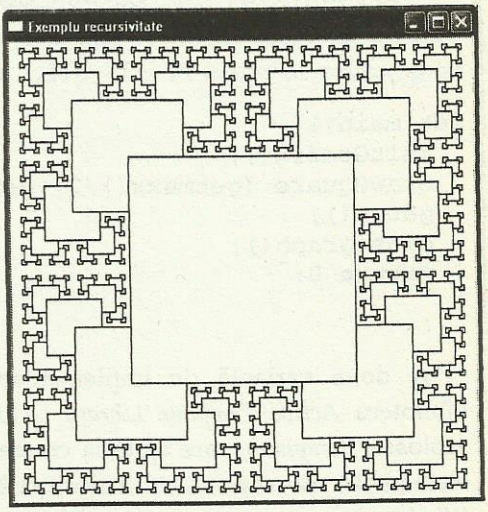
```
int main(){
  int k, x0, y0, n;
  std::ifstream in("quadrate.in");
  std::ofstream out("quadrate.out");
  while(in && !in.eof() && in>>k>>x0>>y0){
    n=0;
    count(x0, y0, 1024, 1024, k, n);
    out<<n<<std::endl;
  }
  return 0;
}
```

Probleme propuse

1. Scrieți și o variantă iterativă ce rezolvă problema dată.
2. Calculați suma ariilor tuturor pătratelor în care se află punctul dat.

Problema 10. Pătrate (recursivitate directă)

Scrieți o aplicație recursivă care desenează figura alăturată.



Analiza problemei și proiectarea soluției

În vârfurile unui pătrat cu lungimea laturii L se desenează câte un pătrat cu lungimea laturii $L/2$. Aceste patru pătrate generate au centrele în vârfurile pătratului inițial. Pentru toate cele patru pătrate astfel desenate, se vor genera mai departe 16 pătrate cu lungimea laturii $L/4$ și centrele în vârfurile lor. Procedeul se repetă până când lungimea laturii pătratelor generate este mai mică decât 10 pixeli. Pentru obținerea figurii de mai sus, avem nevoie de o bibliotecă grafică C++, care, de obicei, este implementată diferit pe sisteme de operare diferite. Vom prezenta două variante de implementare. Prima este simplă și ușor de înțeles, folosind compilatorul Borland C++ 3.1.

Program 1 (Borland C++ 3.1)

```
#include <graphics.h>
#include <iostream.h>
#include <conio.h>
#include <stdlib.h>
```

```
void initGrafix()
{
  int gdet = DETECT, gm, err;
  initgraph(&gdet, &gm, "c:\\borlandc\\BGI");
  err = graphresult();
  if(err != grOk){
    cout<< "Grafic error: "<< grapherrormsg(err)<< endl;
    cout<< "Press any key to quit.";
    getch(); exit(1);
  }
}

void drawSquare(int x, int y, int l)
{
  if(l>50){
   setcolor(14);
   drawSquare(x-l/2, y-l/2, l/2);
   drawSquare(x-l/2, y+l/2, l/2);
   drawSquare(x+l/2, y-l/2, l/2);
   drawSquare(x+l/2, y+l/2, l/2);
   rectangle(x-l/2, y-l/2, x+l/2, y+l/2);
  }
}

int main(){
  initGrafix();
  drawSquare (getmaxx()/2, getmaxy()/2, getmaxy()/2);
  getch();
  closegraph();
  return 0;
}
```

A doua variantă de implementare este una pentru *Windows*, care foloseşte biblioteca *Active Template Library* (*ATL*). *ATL* este o bibliotecă de clase bazată pe şabloane (*template*), care ajută la crearea şi manipularea ferestrelor, a dialogurilor, a elementelor de control etc. Folosirea bibliotecii *ATL* uşurează programarea directă *Windows*, care este mai greoaie, şi foloseşte concepte avansate, cum ar fi moştenirea.

Folosirea ferestrelor în *ATL* se bazează pe clasa *CWindow*, care conţine un *Windows-handle* (*HWND*) şi pe aproape toate metodele *User32* care aşteaptă ca prim parametru *HWND*. Metoda specifică *Windows*, *BOOL ShowWindow (HWND hWnd, int nCmdShow)*, are în *ATL* forma: *BOOL CWindow::ShowWindow (int nCmdShow)*.

ATL oferă mai multe rafinări ale elementelor *CWindow*:

- *CWindowImpl* – înregistrează clasa corespunzătoare ferestrei şi *message handling*;
- *CDialogImpl* – oferă posibilitatea de a crea ferestre de dialog modale şi nemodale;

- *CSimpleDialog* şi *CDialogImpl* – specializări ale dialogurilor modale, ce conţin funcţionalităţi de bază.

În afara claselor corespunzătoare ferestrelor, există şi clase ajutătoare, ce uşurează implementarea obiectelor *ATL* de tip fereastră:

- *CWndClassInfo* – oferă metode de înregistrare pentru informaţiile unei ferestre;
- *CWinTraits* (sau *CWinTraitsOR*) – oferă metode simple pentru standardizarea caracteristicilor unui obiect de tip fereastră din *ATL*.

Pentru a putea folosi *ATL*, trebuie incluse antetele *atlwin.h* şi *atlbase.h*. Acestea ar trebui să fie incluse în *header*-ul (antetul) precompilat.

Pentru versiunile *ATL* mai vechi de 7.0 este necesară declararea unei instanţe *CComModule*. Ea acţionează ca o instanţă globală, al cărei scop principal este acceptarea unei instanţe *HINSTANCE* pentru aplicaţie. Aplicaţia trebuie să apeleze la început *_Module.Init(...)* pentru a crea o instanţă *HINSTANCE* pentru *DLLMain* sau *WinMain*. La sfârşit, trebuie apelată *_Module.Term()* pentru a elibera seturile de date.

Folosirea lui *CComModule* pentru *ATL* 7.0 este depăşită (*obsolete*).

Dacă programul se compilează cu versiuni *ATL* mai vechi (de exemplu, *ATL* 3.0 din *Visual C++ 6.0*), se poate utiliza macroul *_ATL_VER* în *header*-ul precompilat *stdafx.h* cu forma:

```
...
#include <atlbase.h>

#if _ATL_VER < 0x0700
#extern CComModule _Module
#endif

#include <atlwin.h>
...
```

În acest fel, trebuie procedat cu *_Module.Init(...)* şi *_Module.Term(...)* din *WinMain*.

Prezentăm în continuare o a doua soluţie pentru problema dată, folosind *ATL*: antetul precompilat *stdafx.h*.

```
#ifndef _STDAFX_H____
#define _STDAFX_H____

#include <atlbase.h>
#include <atlwin.h>
#endif
```

Stabileşte folosirea *ATL*.

Antetul *MainWindow.h* defineşte în clasa abstractă *CMainWindow* fereastra principală a aplicației şi declară o funcție de tip *factory* (*CreateMainWindow()*) pentru implementările obiectului *CMainWindow*.

Fereastra principală este o implementare *WindowImpl*, cu ajutorul căreia se înregistrează clasele ferestrei şi manipularea mesajelor (*message handling*). Pentru a prelucra mesajele ferestrei, implementăm *ProcessWindowMessage*. Fereastra trebuie să deseneze şi operația respectivă este declarată abstract. Moştenitorii acestei clase oferă posibilitatea implementării operațiilor de desenare.

```
#ifndef _CMAINWINDOW_H_
#define _CMAINWINDOW_H_

class CMainWindow: public CWindowImpl<CMainWindow>{
public:

  virtual BOOL ProcessWindowMessage
    ( HWND hWnd, UINT uMsg,
      WPARAM wParam, LPARAM lParam,
         LRESULT& lResult,
      DWORD dwMsgMapID){
         BOOL bHandled = TRUE;
         switch(uMsg) {

         case WM_PAINT:
           lResult = OnPaint();
           break;

         case WM_DESTROY:
          lResult = OnDestroy();
          break;
        default:
          bHandled = FALSE;
        }
      return bHandled;
  }


protected:

  virtual void OnPaintImpl(HDC &hdc)=0;
```

Este apelată atunci când se primeşte un mesaj pentru fereastră şi returnează o valoare booleană, care specifică dacă mesajul a fost procesat. Implementarea reacționează la două feluri de mesaje: *WM_PAINT* şi *WM_DESTROY*.

Sistemul sau alt program cer desenarea unei ferestre de aplicație.

Fereastra va fi distrusă.

Alte mesaje rămân neprocesate.

Desenarea este o metodă abstractă.

```
ivate:
      LRESULT OnPaint() {
            PAINTSTRUCT ps;
            HDC hdc=BeginPaint(&ps);
            OnPaintImpl(hdc);
            EndPaint(&ps);
            return 0;
      };

      LRESULT OnDestroy() {

            PostQuitMessage(0);
            return 0;
      }


ainWindow *CreateMainWindow();

ndif
```

Pregăteşte fereastra pentru desenare, apelează metoda de desenare şi marchează sfârşitul acesteia.

Semnalizează sistemului faptul că *thread*-ul va trebui părăsit. Bucla cu mesaje se va încheia şi va transmite controlul înapoi sistemului.

Declararea funcţiei de tip *factory* pentru implementarea *CMainWindow*.

ainWindowImpl.cpp – implementarea clasei *CMainWindow*, are desenează pătrate:

```
#include "stdafx.h"
#include "MainWindow.h"

lass PaintInvariant {
  public:
      PaintInvariant(HDC& refHdc, int minimumL = 10):
            rHdc(refHdc),dx(0),dy(0),minL(minimumL) {
      }

      HDC& rHdc;

      int dx;

      int dy;
```

Grupează parametrii ce rămân constanţi în timpul desenării.

Referinţă la un *device context* – un set pentru desenarea obiectelor grafice.

Spaţiul dintre conturul stâng/drept al ferestrei şi suprafaţa pătratului, care trebuie redesenată.

Spaţiul dintre conturul sus/jos al ferestrei şi suprafaţa pătratului, care trebuie redesenată.

```
        int minL;
};

class CQuadratWindow: public CMainWindow
{
  protected:
        virtual void OnPaintImpl(HDC &hdc);
  private:
        void _drawRec(PaintInvariant &rPaintInvariant,
                      int x, int y, int l);
};

CMainWindow *CreateMainWindow() {
        return new CQuadratWindow();
}

void CQuadratWindow::OnPaintImpl(HDC &hdc){

        RECT rect;
        GetClientRect(&rect);

        int diff = rect.right-rect.bottom;
        PaintInvariant paintInv(hdc,5);
        int l;
        if(diff<0){
              diff *=-1;
              l = rect.right/2;
              paintInv.dx = 0;
              paintInv.dy = diff/2;
        }
        else{
              l = rect.bottom/2;
              paintInv.dx = diff/2;
              paintInv.dy=0;
```

Lungimea minimă a laturii unui pătrat.

Implementarea *CMainWindow*, care desenează pătrate.

Desenează pătrate.

Implementarea funcției de tip *factory* pentru *CMainWindow*. Furnizează o nouă instanță a clasei *CQuadratWindow*, care este o funcție declarată în *header*-ul *MainWindow.h*.

Implementarea metodei de desenare.

Stabilește zona în care se poate desena.

Inițializează datele invariante (variabilele care nu se vor schimba în timpul desemnării).

```
    }

    _drawRec(paintInv, 1, 1, 1);


void CQuadratWindow::_drawRec(PaintInvariant &rPaintInvariant,
                              int x, int y, int l){

    if(l>rPaintInvariant.minL) {

        _drawRec(rPaintInvariant, x-l/2, y-l/2, l/2);
        _drawRec(rPaintInvariant, x-l/2, y+l/2, l/2);
        _drawRec(rPaintInvariant, x+l/2, y-l/2, l/2);
        _drawRec(rPaintInvariant, x+l/2, y+l/2, l/2);
        Rectangle(rPaintInvariant.rHdc,
              x-l/2  + rPaintInvariant.dx,
              y-l/2  + rPaintInvariant.dy,
              x+l/2  + rPaintInvariant.dx,
              y+l/2  + rPaintInvariant.dy);

    }
```

Desenează un pătrat.

main.cpp:

```
#include "stdafx.h"
#include "MainWindow.h"

#include <memory>
```

Pentru *std::auto_ptr – „smart pointer"*

```
int APIENTRY _tWinMain (HINSTANCE hinst,
                        HINSTANCE hPrevInstance,
                        LPTSTR lpCmdLine,
                        int nCmdShow){
```

Apelează *CreateMainWindow()* și salvează *pointer*-ul obiectului ferestrei principale într-un *smart pointer*. Folosim acest tip de *pointer* pentru că, atunci când *auto_pointer* iese din scopul său, destructorul ferestrei șterge *pointer*-ul la fereastră.

```
std::auto_ptr<CMainWindow> pWndAuto(CreateMainWindow());
```

Atunci când *CreateMainWindow()* nu poate să creeze o fereastră (de exemplu, deoarece nu există destulă memorie), se va returna NULL. În acest caz, aplicaţia se va termina.

```
if(!pWndAuto.get()) {
  return -1;
}
```

Creează fereastra: mai întâi se înregistrează clasa ferestrei, dacă aceasta nu s-a întâmplat până acum. Noua fereastră creată se va ataşa automat obiectului de tip fereastră.

```
pWndAuto->Create
    (0
     ,CWindow::rcDefault,
    __T("Exemplu recursivitate"),
    WS_OVERLAPPEDWINDOW,
    WS_EX_CLIENTEDGE
    );
```

Nu există ferestre-părinte.

Dimensiunile *default* ale ferestrei.

Numele ferestrei.

Stilul ferestrei.

Stil avansat al ferestrei.

```
pWndAuto->CenterWindow();
```

Centrează fereastra pe ecran.

```
pWndAuto->ShowWindow(nCmdShow);
```

Desenează fereastra.

Actualizează aria client prin trimiterea unui mesaj *WM_PAINT* ferestrei, atunci când zona ferestrei nu este goală.

```
pWndAuto->UpdateWindow();
```

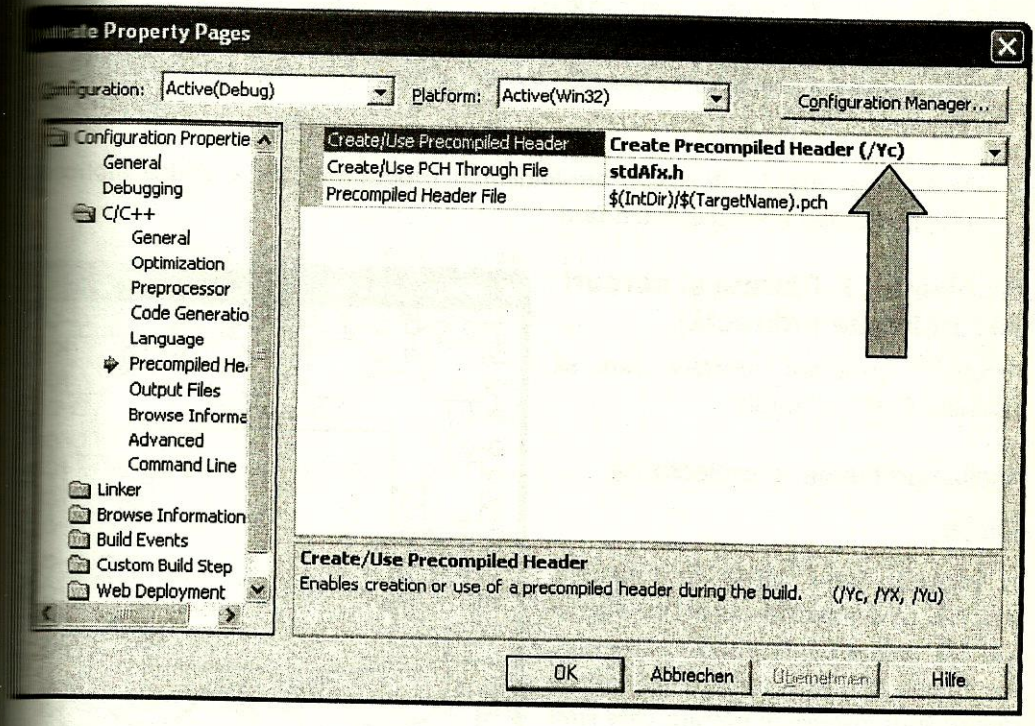
Bucla principală de mesaje: ea va fi părăsită atunci când se primeşte mesajul *WM_DESTROY*.

```
MSG msg;
while (GetMessage(&msg,0,0,0)){
  TranslateMessage(&msg);
  DispatchMessage(&msg);
}
```

```
return (int) msg.wParam;
```

Codul de ieşire este valoarea furnizată de către *PostQuitMessage*: în cazul nostru, 0.

Observaţie. Trebuie să aveţi în vedere faptul că, în meniul cu proprietăţile proiectului, secţiunea *Precompiled Header* se va alege opţiunea *Create Precompiled Header*, ca în figură:



Meniul de proprietăţi al proiectului

Probleme propuse

1. Modificaţi programul astfel încât să se deseneze figura 1 (pătratele mai mici nu mai trebuie acoperite de cele mai mari).
2. Modificaţi programul, astfel încât, în loc de pătrate, să se deseneze triunghiuri echilaterale (figura 2). Centrat în fiecare vârf al triunghiului se desenează un alt triunghi echilateral, a cărui latură are lungimea jumătate din latura triunghiului iniţial.

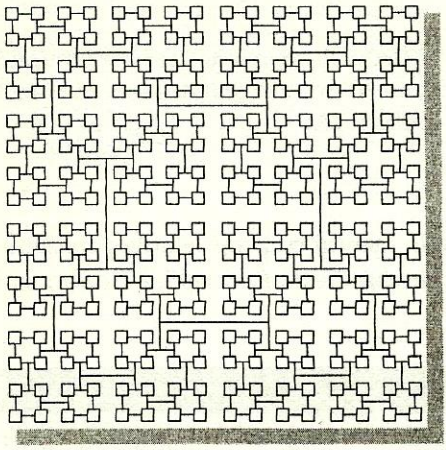


Figura 1

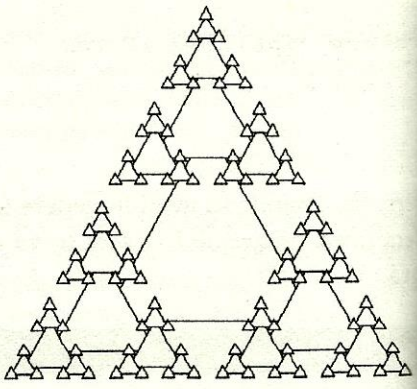
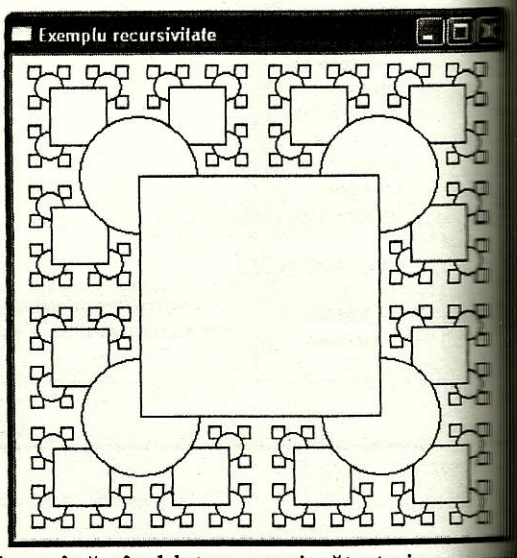


Figura 2

3. Dacă programați în *Windows*, implementați programul fără a utiliza bibliotec *ATL*, ci facilitățile *Windows* standard.

Problema 11. Pătrate şi cercuri (recursivitate indirectă)



Scrieți un program recursiv, care să deseneze figura alăturată.

Analiza problemei şi proiectarea soluției

În vârfurile unui pătrat cu lungimea laturii *L*, se desenează cercuri cu diametrul *L*/2, centrate în vârfurile pătratului. Ne imaginăm mai departe că aceste cercuri sunt înscrise în pătrate, care sunt invizibile şi în vârfurile lor se vor desena corespunzător pătrate centrate cu laturile înjumătățite. Procesul se va relua până când latura unui pătrat ajunge mai mică decât 10. Prezentăm mai jos o implementare *DOS* simplă, care este compilată cu *Borland C++ 3.1*:

Program

```
#include <graphics.h>
#include <iostream.h>
#include <stdlib.h>

void initGrafix()

 int gdet = DETECT, gm, err;
 initgraph(&gdet, &gm, "c:\\borlandc\\BGI");
 err = graphresult();
 if(err != grOk)
 {
   cout<< "Grafic error: "<< grapherrormsg(err)<< endl;
   cout<< "Press any key to quit.";
   exit(1);

 }


void drawSquare(int x, int y, int l);

void drawCircle(int x, int y, int r)

 if(r>25)
 {
   circle(x, y, r);
   drawSquare(x, y, r);
 }


void drawSquare (int x, int y, int l)

 if(l>10)
 {
  rectangle(x-l/2, y-l/2, x+l/2, y+l/2);
  drawCircle (x-l/2, y-l/2, l/2);
  drawCircle (x-l/2, y+l/2, l/2);
  drawCircle (x+l/2, y-l/2, l/2);
  drawCircle (x+l/2, y+l/2, l/2);
 }


int main()

 initGrafix();
 setcolor(0);
 setlinestyle(0, 1, 2);
```

```
  setfillstyle(1, 15);
  bar(0, 0, getmaxx(), getmaxy());
  drawSquare(getmaxx()/2, getmaxy()/2, getmaxy()/2);
  closegraph();
  return 0;
}
```

Prezentăm în continuare o implementare *Windows*, care seamănă cu implementarea *ATL* de la problema anterioară. Se creează un proiect cu fişierele *stdafx.h*, *MainWindow.h* şi *main.cpp*, care coincid cu cele de mai sus. Ceea ce se schimbă este doar implementarea *MainWindowImpl.cpp*, după cum urmează:

```
#include "stdafx.h"
#include "MainWindow.h"

class PaintInvariant{
public:
  PaintInvariant(HDC& refHdc, int minimumL = 10):
      rHdc(refHdc), dx(0), dy(0), minL(minimumL) {}

    HDC& rHdc;

    int dx;

    int dy;

    int minL;
  };

class CSquareCircleWindow: public CMainWindow{
  protected:

    virtual void OnPaintImpl(HDC &hdc);

  private:

    void _drawRec(PaintInvariant &rPaintInvariant,
         int x, int y, int l);
    void _drawCircle(PaintInvariant &rPaintInvariant,
         int x, int y, int l);
};

CMainWindow *CreateMainWindow() {
  return new CSquareCircleWindow();
}
```

```
void CSquareCircleWindow::OnPaintImpl(HDC &hdc) {

  RECT rect;
  GetClientRect(&rect);

  int diff = rect.right-rect.bottom;
  PaintInvariant paintInv(hdc, 10);
  int l;
  if(diff<0) {
    diff *=-1;
    l = rect.right/2;
    paintInv.dx = 0;
    paintInv.dy = diff/2;
    }
  else{
    l = rect.bottom/2;
    paintInv.dx = diff/2;
    paintInv.dy=0;
    }

  _drawRec(paintInv, l, l, l);
}

void CSquareCircleWindow::_drawRec(
                        PaintInvariant &rPaintInvariant,
                        int x, int y, int l)
{
  if(l>rPaintInvariant.minL) {
    _drawCircle(rPaintInvariant, x-l/2, y-l/2, l/2);
    _drawCircle(rPaintInvariant, x-l/2, y+l/2, l/2);
    _drawCircle(rPaintInvariant, x+l/2, y-l/2, l/2);
    _drawCircle(rPaintInvariant, x+l/2, y+l/2, l/2);
    Rectangle(rPaintInvariant.rHdc,
      x-l/2  + rPaintInvariant.dx,
      y-l/2  + rPaintInvariant.dy,
      x+l/2  + rPaintInvariant.dx,
      y+l/2  + rPaintInvariant.dy);
    }
}

void CSquareCircleWindow::_drawCircle(
                        PaintInvariant &rPaintInvariant,
                        int x, int y, int l)
{
  if(l>rPaintInvariant.minL) {
    _drawRec(rPaintInvariant, x-l/2, y-l/2, l/2);
    _drawRec(rPaintInvariant, x-l/2, y+l/2, l/2);
```

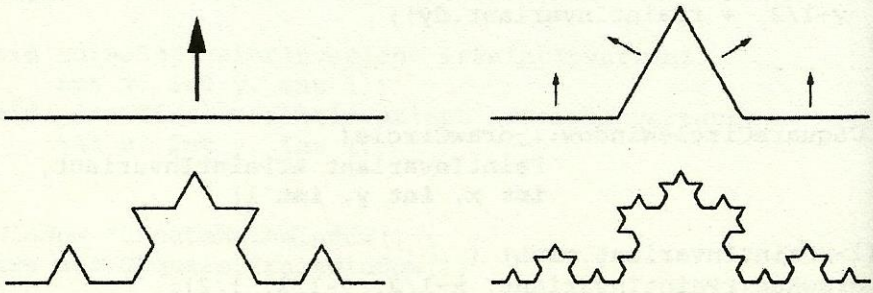
```
    _drawRec(rPaintInvariant, x+l/2, y-l/2, l/2);
    _drawRec(rPaintInvariant, x+l/2, y+l/2, l/2);
    Ellipse(rPaintInvariant.rHdc,
      x-l/2  + rPaintInvariant.dx,
      y-l/2  + rPaintInvariant.dy,
      x+l/2  + rPaintInvariant.dx,
      y+l/2  + rPaintInvariant.dy);
    }
}
```

Probleme propuse

1. Calculaţi suma ariilor vizibile ale tuturor cercurilor.
2. Implementaţi un program care foloseşte de asemenea recursivitatea indirectă astfel:
   - *drawSquare* apelează de patru ori metoda *drawCircle* (desenează patru cercuri de diametru *L*/2, ale căror centre sunt vârfurile pătratului);
   - *drawCircle* apelează o dată *drawTriangle* (desenează triunghiul echilateral circumscris cercului);
   - *drawTriangle* apelează de trei ori *drawSquare* (desenează trei pătrate cu latura jumătate din cea a triunghiului, centrate în vârfurile sale).

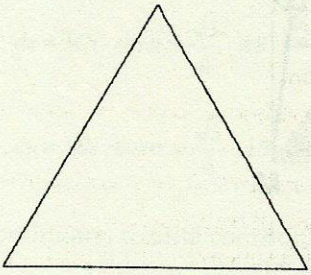
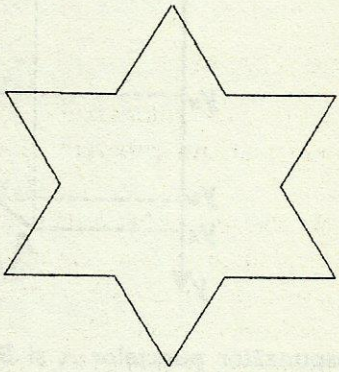
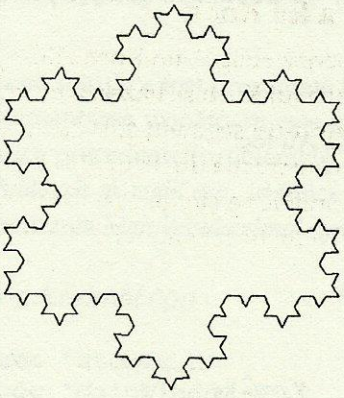
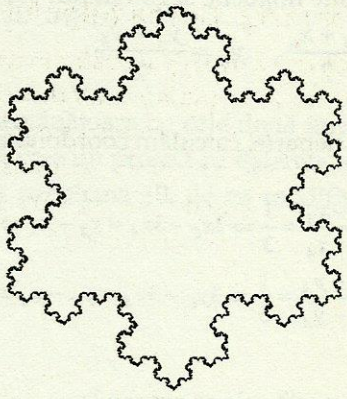
Problema 12. Curba fulgului de zăpadă a lui Koch

A fost descrisă în 1904 de către matematicianul suedez Helge von Koch (1870-1924) şi este unul dintre primii fractali descrişi formal. Curba lui Koch se bazează pe un proces construit iterativ: la început, este formată numai dintr-un segment şi la fiecare iteraţie fiecare segment al curbei se transformă prin „construcţia în triunghi echilateral" (segmentul se împarte în trei părţi egale, cea din mijloc se şterge şi se înlocuieşte cu alte două segmente, ce formează cu cea ştearsă un triunghi echilateral).

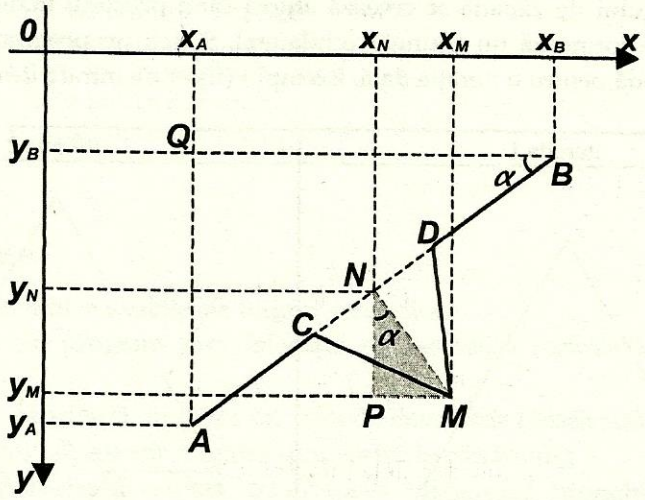


Algoritmul de obţinere a curbei lui Koch

Curba fulgului de zăpadă se creează atunci când procesul iterativ începe cu trei segmente care formează un triunghi echilateral. Scrieți un program care desenează fulgul de zăpadă pentru o iterație dată. Exemplu (fișier de intrare *iteration.in*):

Iterația 1	Iterația 2
	
Iterația 4	**Iterația 6**
	

Analiza problemei şi proiectarea soluției



Corespunzător punctelor A şi B în plan, cu coordonatele $A(x_A, y_A)$ şi $B(x_B, y_B)$, trebuie să găsim coordonatele punctelor C, D şi M cu proprietățile: $C, D \in$ dreptei (AB), $AC \equiv CD \equiv DB \equiv CM \equiv MD$.

Notăm mijlocul segmentului AB cu $N(x_N, y_N)$. Coordonatele acestui punct sunt:

$$x_N = \frac{x_A + x_B}{2}, \quad y_N = \frac{y_A + y_B}{2}.$$

Mai departe, calculăm coordonatele punctului C. Au loc:

$$\frac{x_C - x_A}{x_B - x_A} = \frac{1}{3} \Rightarrow 3x_C - 3x_A = x_B - x_A \Rightarrow x_C = \frac{2x_A + x_B}{3},$$

$$\frac{y_C - y_A}{y_B - y_A} = \frac{1}{3} \Rightarrow 3y_C - 3y_A = y_B - y_A \Rightarrow y_C = \frac{2y_A + y_B}{3}.$$

Pe această cale se ajunge la: $x_D = \frac{x_A + 2x_B}{3}$ şi $y_D = \frac{y_A + 2y_B}{3}$.

Deoarece MCD este un triunghi echilateral cu lungimea laturii $\frac{AB}{3}$ şi înălțimea MN, rezultă:

$$MN = CM \cdot \sin\frac{\pi}{3} = \frac{AB}{3} \cdot \frac{\sqrt{3}}{2} = \frac{\sqrt{3}}{6} \cdot AB.$$

Observăm că $\alpha = \sphericalangle PNM = \sphericalangle QBA$ (*PN* este perpendiculară pe *QB* şi *NM* perpendiculară pe *AB*: $PN \perp QB$ şi $NM \perp AB$). În triunghiul *ABQ* au loc:

$$\sin\alpha = \frac{y_A - y_B}{AB} \text{ şi } \cos\alpha = \frac{x_B - x_A}{AB}.$$

Şi le folosim în triunghiul *NMP*:

$$dx = MN \cdot \sin\alpha = \frac{\sqrt{3}}{6} \cdot AB \cdot \frac{y_A - y_B}{AB} = \frac{\sqrt{3}}{6} \cdot (y_A - y_B)$$

$$dy = MN \cdot \cos\alpha = \frac{\sqrt{3}}{6} \cdot AB \cdot \frac{x_B - x_A}{AB} = \frac{\sqrt{3}}{6} \cdot (x_B - x_A).$$

În continuare, rezultă coordonatele punctului *M*:

$$x_M = x_N + dx = \frac{x_A + x_B}{2} + \frac{\sqrt{3}}{6} \cdot (y_A - y_B)$$

$$y_M = y_N + dy = \frac{y_A + y_B}{2} + \frac{\sqrt{3}}{6} \cdot (x_B - x_A).$$

Această soluţie a fost prezentată în [Ran99].

Implementarea următoare este în *Windows*, asemănătoare cu cele două implementări grafice precedente (problemele 10 şi 11). Se va crea un proiect cu fişierele *stdafx.h*, *MainWindow.h* şi *main.cpp*, identice cu cele de la problema 10. Se va modifica doar implementarea *MainWindowImpl.cpp*, ca mai jos.

MainWindowImpl.cpp

```
#include "stdafx.h"
#include "MainWindow.h"

#include <fstream>
#include <cmath>

class PaintInvariant {
public:
  PaintInvariant(HDC& refHdc, int minimumL = 10):
      rHdc(refHdc),dx(0),dy(0), minL(minimumL) {}
```

```
    HDC& rHdc;
    int dx;
    int dy;
    int minL;
};

class CKochWindow: public CMainWindow {
  protected:
    virtual void OnPaintImpl(HDC &hdc);
  private:
    void _drawKoch(PaintInvariant &rPaintInvariant, int n,
      int xa, int ya, int xb, int yb);
};

CMainWindow *CreateMainWindow() {
  return new CKochWindow();
}

void CKochWindow::OnPaintImpl(HDC &hdc) {

  RECT rect;
  GetClientRect(&rect);

  int diff = rect.right-rect.bottom;
  PaintInvariant paintInv(hdc, 5);
  int l;
  if(diff<0) {
    diff *=-1;
    l = rect.right/2;
    paintInv.dx = 0;
    paintInv.dy = diff/2;
  }
  else{
    l = rect.bottom/2;
    paintInv.dx = diff/2;
    paintInv.dy=0;
  }

  int n;
  std::ifstream in("iteration.in");
  if(in && !in.eof()){
    in>>n;
    double xa, ya, xb, yb, t=l;
    double aux=sqrt(3.0);
    xa=l; ya=l-t;
    xb=l-aux*t/2; yb=l+t/2;
    _drawKoch(paintInv,  n,
              (int)xa, (int)ya, (int)xb, (int)yb);
```

```
    xa=l+aux*t/2; ya=l+t/2;
    _drawKoch(paintInv,  n,
              (int)xb, (int)yb, (int)xa, (int)ya);
    xb=l; yb=l-t;
    _drawKoch(paintInv, n,
              (int)xa, (int)ya, (int)xb, (int)yb);
}

void CKochWindow::_drawKoch(PaintInvariant &rPaintInvariant,
                    int n, int xa, int ya, int xb, int yb){
double r=sqrt(3.0)/6.0;
if(n>1){

   double xc, yc, xm, ym, xd, yd;
   xm=(xa+xb)/2+r*(ya-yb);
   ym=(ya+yb)/2+r*(xb-xa);
   xc=(2*xa+xb)/3; yc=(2*ya+yb)/3;
   xd=(xa+2*xb)/3; yd=(ya+2*yb)/3;


   _drawKoch(rPaintInvariant, n-1,
             (int)xa, (int)ya, (int)xc, (int)yc);
   _drawKoch(rPaintInvariant, n-1,
             (int)xc, (int)yc, (int)xm, (int)ym);
   _drawKoch(rPaintInvariant, n-1,
             (int)xm, (int)ym, (int)xd, (int)yd);
   _drawKoch(rPaintInvariant, n-1,
             (int)xd, (int)yd, (int)xb, (int)yb);
 } else{

    POINT p[2];
    p[0].x=xa; p[0].y=ya;
    p[1].x=xb; p[1].y=yb;

    Polygon(rPaintInvariant.rHdc, p, 2);
}
```

$$x_M = \frac{x_A + x_B}{2} + \frac{\sqrt{3}}{6} \cdot (y_A - y_B)$$

$$y_M = \frac{y_A + y_B}{2} + \frac{\sqrt{3}}{6} \cdot (x_B - x_A)$$

$$x_C = \frac{2x_A + x_B}{3},\ y_C = \frac{2y_A + y_B}{3}$$

$$x_D = \frac{x_A + 2x_B}{3},\ y_D = \frac{y_A + 2y_B}{3}$$

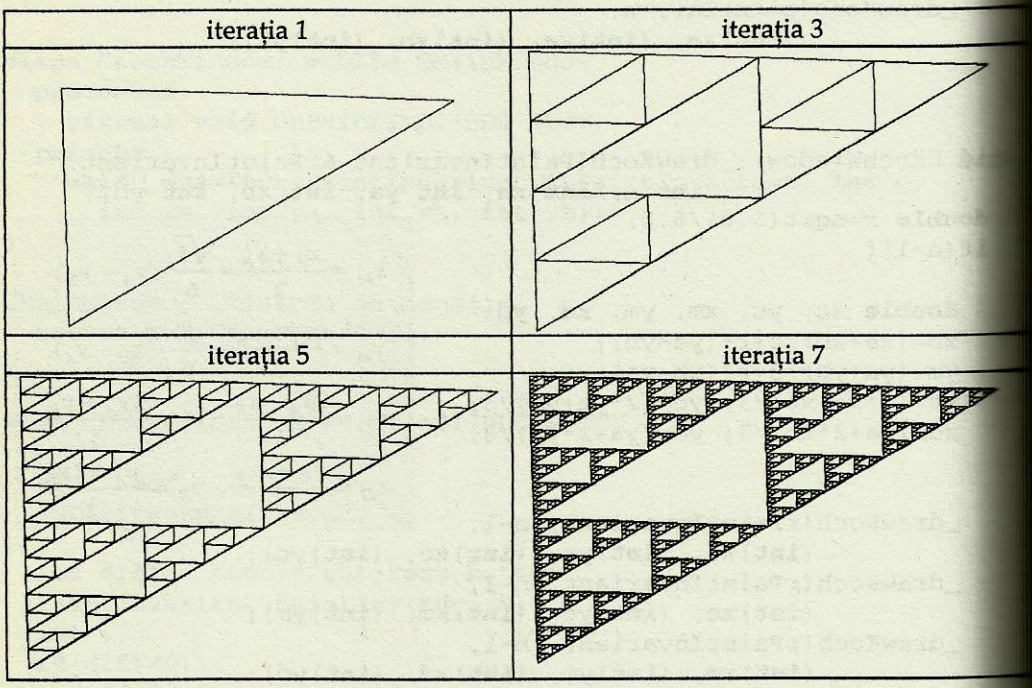
Desenează segmentul *AB*. Altă posibilitate:

```
MoveToEx(rPaintInvariant.rHdc,
         xa, ya, NULL);
LineTo(rPaintInvariant.rHdc, xb, yb);
```

Probleme propuse

1. Calculați aria pe care o ocupă curba fulgului de zăpadă la o iterație dată atunci când triunghiul inițial are lungimea laturii *L*. Care sunt lungimea curbei și aria pentru număr de iterații care tinde la infinit?
2. *Triunghiul lui Sierpinski.* Se consideră un triunghi oarecare, în care se trasează cele trei linii mijlocii și se elimină triunghiul care se formează în interior. Acest

procedeu se repetă pentru triunghiurile formate, iar figura obținută poartă numele de *triunghiul lui Sierpinski*. Scrieți un program care desenează triunghiul lui Sierpinski pentru o iterație dată. Exemplu:



Calculați aria triunghiului lui Sierpinski pentru o iterație dată, atunci când triunghiul inițial are aria A. Determinați această arie atunci când numărul iterației tinde la infinit.

3. *Fractal space-filling*. Fie mulțimea $P(0, 1) = \{(x, y) \mid 0 < x, y < 1\}$, care este mulțimea tuturor punctelor din pătratul unitate cu vârfurile (0, 0), (0, 1), (1, 1) și (1, 0). În [Bar93] este prezentată și demonstrată o metodă care „umple" acest pătrat. Se consideră următoarele tablouri:

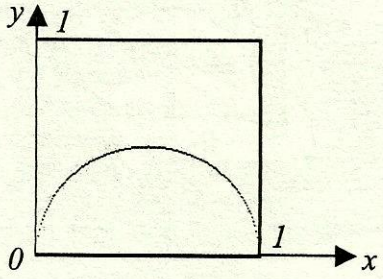
Index	*a*	*b*	*c*	*d*	*e*	*f*
1	0	0.5	0.5	0	0	0
2	0.5	0	0	0.5	0	0.5
3	0.5	0	0	0.5	0.5	0.5
4	0	-0.5	-0.5	0	1	0.5

Fie K_0 o curbă în $P(0, 1)$. Pentru toate numerele naturale i $\alpha \geq 0$, se creează curba K_{i+1} pe baza curbei K_i, prin înlocuirea fiecărui punct din K_i cu următoarele patru puncte:

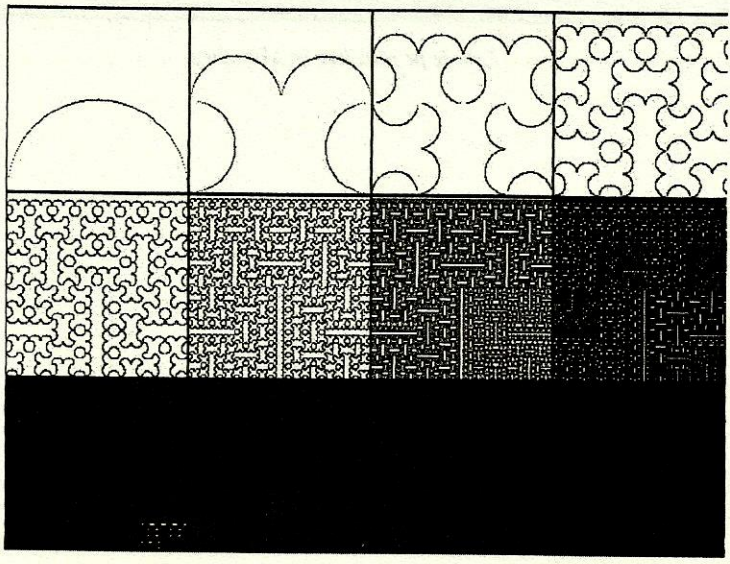
$$w_n\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a_n & b_n \\ c_n & d_n \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} e_n \\ f_n \end{bmatrix} \text{ pentru } n = 1, \ldots, 4.$$

Au loc deci: $\begin{cases} x_{n,i+1} = a_n \cdot x_i + b_n \cdot y_i + e_n \\ y_{n,i+1} = c_n \cdot x_i + d_n \cdot y_i + f_n \end{cases}$, pentru $n = 1, \ldots, 4$. Formal, putem scrie pentru orice $i \geq 0$: $K_{i+1} = \bigcup_{n=1}^{4} \{(x_{n,i+1}, y_{n,i+1}) | (x_i, y_i) \in K_i\}$.

Pentru curba inițială K_0:

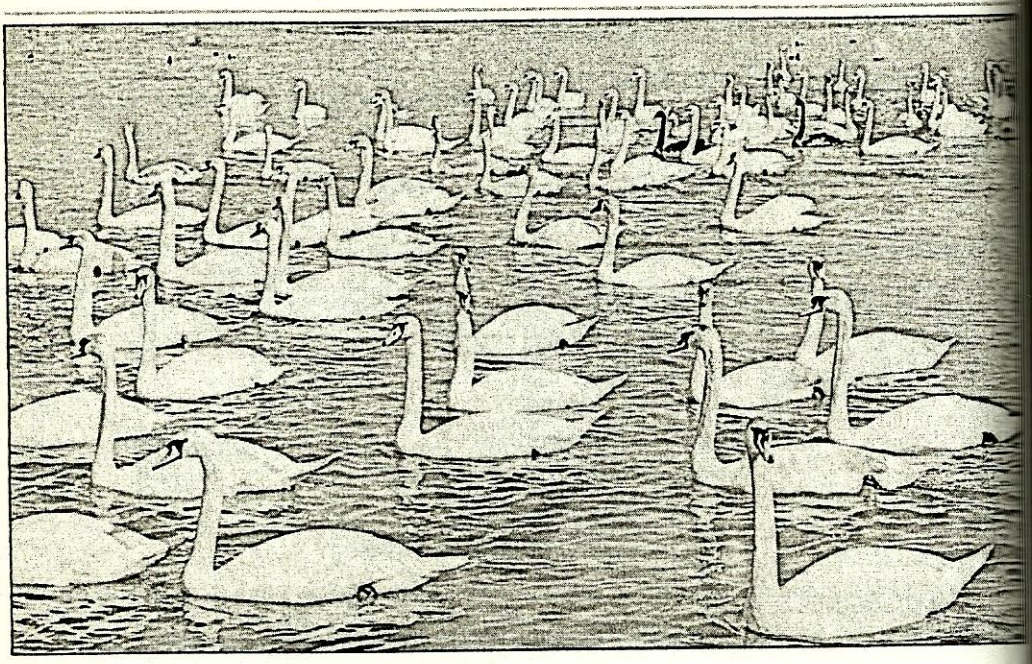


se obțin K_0, K_1, ..., K_{11}:



Scrieți un program care creează diverse curbe inițiale K_0 și desenează în această formă primele 12 iterații.

Notă: o rezolvare în *Borland C* pentru *DOS* a problemei, folosind operatorii pe biți, se găsește în [Log06], p. 394.



Lebede pe râul Isar în München

Capitolul 9
Divide et Impera

7 probleme complet rezolvate, 18 probleme propuse

- Fundamente
- Cel mai mare divizor comun
- Turnurile din Hanoi
- Integrală cu regula trapezului
- *QuickSort* (sortarea rapidă)
- *MergeSort* (sortarea prin interclasare)
- Arbori *Quad* (*Quad-Trees*)
- Transformata *Fourier* discretă (*DFT*)

O fi bine, o fi rău, uneori este foarte plăcut să spargi ceva.
F.M. Dostoievski

Fundamente

Se presupune că formularea *divide et impera* îi aparţine regelui francez Ludovic al XI-lea (1461-1483). Unii istorici însă i-o atribuie şi lui Iulius Cezar (100 î.Ch.-44 î.Ch.). Ea ar trebui să însemne că adversarii trebuie împrăştiaţi şi slăbiţi prin lupte între ei, pentru a putea fi învinşi şi supuşi.

Vom considera mai departe însă această formulare dintr-o perspectivă mult mai civilizată. Îndemnul trebuie să ne ajute la soluţionarea unor probleme prin împărţirea lor în subprobleme de dimensiune mai mică, rezolvarea lor şi combinarea rezultatelor în scopul obţinerii soluţiei pentru problema iniţială. La rândul lor, subproblemele se pot împărţi în probleme asemănătoare de dimensiuni mai mici, care se rezolvă cu aceeaşi metodă. Această manevră se reia recursiv, până când subproblemele obţinute devin elementare şi pot fi rezolvate direct.

Paşii rezolvării unei probleme folosind principiul *Divide et Impera* sunt:

- *Divide*: împarte problema recursiv în subprobleme, până când au dimensiuni elementare;
- *Impera*: rezolvă subproblemele;
- *Impera*: combină soluţiile subproblemelor pentru obţinerea soluţiei problemei iniţiale.

Forma generală a algoritmului:

```
ALGORITM_DivideEtImpera(P, n, S)
  If (n≤n0) Then
    directSolve(P, n, S)
  End_If
  Else Execute
    Divide (P, n) in (P1, n1), (P2, n2), ..., (Pk, nk)
    DivideEtImpera(P1, n1, S1)
    DivideEtImpera(P2, n2, S2)
    ...
    DivideEtImpera(Pk, nk, Sk)
    Combine S1, S2, ..., Sk to obtain S
  End_Else
END_ ALGORITM_DivideEtImpera(P, n, S)
```

Exemple cunoscute pentru aplicarea metodei sunt: problema turnurilor din Hanoi, *QuickSort*, *MergeSort*.

Problema 1. Cel mai mare divizor comun

În fișierul *numere.in* se găsesc mai multe numere naturale (maxim 2000), separate prin spații. Scrieți o metodă care să calculeze cel mai mare divizor comun al acestora. Atunci când fișierul de intrare nu există sau nu conține date, scrieți 0 în fișierul de ieșire.

numere.in	ecran
45 60 125 345 65 9875 4555	cmmdc = 5

Analiza problemei și proiectarea soluției

Definim tipul *TVector* pentru a stoca numerele citite din fișierul de intrare. Este evident că putem calcula cel mai mare divizor comun pentru prima jumătate a vectorului și pentru a doua jumătate, iar soluția problemei este cel mai mare divizor comun al celor două. Condiția de terminare este atinsă atunci când numărul elementelor devine 1 sau 0.

Program

```
#include <fstream>
#include <iostream>
#include <vector>

typedef std::vector<unsigned> TVector;
unsigned cmmdc(unsigned a, unsigned b){
  unsigned r;
  while(b){
    r = a % b;
    a = b;
    b = r;
  }
  return a;
}

unsigned Divide_et_Impera(TVector& a, int iMin, int iMax){
  if(iMin<iMax){
    int middle = (iMin+iMax)/2;
    int d1 = Divide_et_Impera(a, iMin, middle);
    int d2 = Divide_et_Impera(a, middle + 1, iMax);
```

Algoritmul lui Euclid

1. $a_1 \leftarrow a$, $b_1 \leftarrow b$, $i \leftarrow 1$
2. **While** ($b_i \neq 0$) **Do**
   - 2.1. $a_{i+1} \leftarrow b_i$
   - 2.2. $b_{i+1} \leftarrow r_i$ ($=a_i$ **mod** b_i)
   - 2.3. $i \leftarrow i+1$

   End_While
3. cmmdc(a, b) = r_{i-1}

```
    return cmmdc(d1, d2);
  }
  if(0<=iMin && iMin<a.size()) return a[iMin];
   else return 0;
}

int main(){
  std::ifstream f("numere.in");
  unsigned  i, n = 0;
  TVector a;
  while(f && !f.eof() && f>>i) a.push_back(i);
  std::cout << "cmmdc = " <<Divide_et_Impera(a, 0, a.size()-1);
  return 0;
}
```

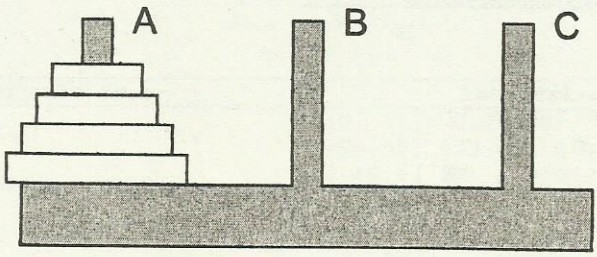
Probleme propuse

1. Scrieți un program care să folosească aceeași metodă pentru a calcula cel mai mare și cel mai mic dintre numerele date.
2. Scrieți un program care, prin aceeași metodă, să caute un număr dat în fișierul *numere.in*.

Problema 2. Turnurile din Hanoi

Se presupune că unul dintre cele mai populare jocuri din toate timpurile, turnurile din Hanoi, a fost inventat în 1883 de către francezul Édouard Lucas. Acesta este unul dintre exemplele standard pentru metoda *Divide et Impera* în literatura de specialitate. Jocul este format din trei tije, *A*, *B* și *C*, pe care se introduc unul sau mai multe discuri. La început, toate discurile se află pe tija *A*, în ordinea descrescătoare a diametrelor, cel mai mare disc fiind la baza tijei. Scopul jocului este mutarea tuturor discurilor de pe tija *A* pe tija *C* astfel încât discurile să fie în aceeași ordine ca pe *A*. Regulile jocului:

- la un moment dat, se mută doar un disc de pe o tijă pe alta;
- nu este permisă amplasarea unui disc de dimensiune mai mare peste un disc de dimensiune mai mică.



Găsiți pentru un număr natural n, citit de la tastatură, cea mai scurtă secvență de mutări ce rezolvă problema pentru n discuri inițiale. Mutările se vor scrie în fișierul *hanoi.out*, ca în exemplu:

Tastatură	hanoi.out
3	(A,C) (A,B) (C,B) (A,C) (B,A) (B,C) (A,C)

Analiza problemei şi proiectarea soluției

Soluția se construiește pornind de la următoarea observație: pentru a muta n discuri de pe tija A pe tija C cu ajutorul tijei B, vom muta mai întâi $n-1$ discuri de pe tija A pe tija B prin intermediul tijei C, apoi executăm mutarea $A \to C$, urmată de mutarea a $n-1$ discuri de pe tija B pe tija C prin intermediul tijei A, adică:

$$Hanoi(n,A,C,B) = \begin{cases} A \to C, \text{ dacă } n=1 \\ Hanoi(n-1,A,B,C), A \to C, Hanoi(n-1,B,C,A), \text{ altfel.} \end{cases}$$

Program

```
#include <fstream>
#include <iostream>

std::ofstream f("hanoi.out");

void write(char A, char B){
  f << "(" << A << ","<< B <<")";
}

void hanoi(int n, char A, char C, char B){
  if (1==n) write(A, C);
  else{
    Hanoi(n-1, A, B, C);
    write(A, C);
    Hanoi(n-1, B, C, A);
  }
}

int main(){
  int n;
  std::cout << " n = ";
  std::cin >> n;
  hanoi(n, 'A', 'C', 'B');
  return 0;
}
```

Probleme propuse

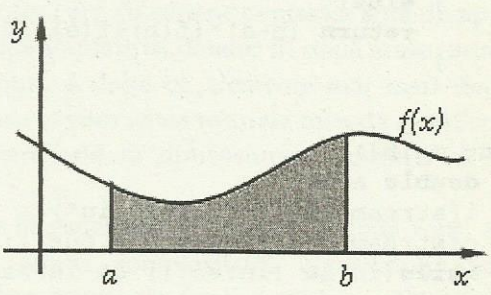
1. Demonstrați că numărul total de mutări este $2^n - 1$.
2. Scrieți un program care simulează grafic într-o formă cât mai frumoasă și mai elocventă mutarea discurilor.
3. Implementați un program nerecursiv pentru rezolvarea problemei: la fiecare mutare impară, se va muta cel mai mic disc (1) de pe tija pe care se află pe următoarea în succesiunea circulară de tije *ABCABCABC...*; pentru mutările pare, se execută mutarea posibilă care nu implică discul 1.
4. Numărul minim de mutări necesare aplicând algoritmul anterior este $2^n - 1$. Pentru două numere naturale n ($0 \le n \le 100$) și m ($0 \le m \le 2^n - 1$), determinați a m-a mutare și configurația tijelor din acel moment.

Problema 3. Integrală cu regula trapezului

Integrala unei funcții în plan este aria suprafeței care se formează între graficul funcției și axa *Ox*. Atunci când funcția are două variabile, integrala reprezintă un volum. Regula trapezului este o metodă matematică cunoscută pentru aproximarea numerică a integralei unei funcții pe intervalul [*a*, *b*] (suprafața gri din figura alăturată). În zona acoperită de funcție în intervalul dat se construiește un trapez și se calculează aria acestuia. Pentru a aproxima mai bine aria, intervalul dat se împarte în mai multe intervale mai mici și aria dintre curba funcției și axa *Ox* se apropie de suma ariilor trapezelor. Calculați valoarea $\int_a^b \frac{1}{1+x^2} dx$ cu ajutorul metodei trapezelor, astfel încât înălțimea fiecărui trapez a cărui arie se sumează să fie mai mică decât *epsilon* = 0,0001.

Date de intrare. În fișierul *integrala.in* se găsesc mai multe perechi (*a*, *b*) de numere reale, astfel încât $-200 \le a$, $b \le 200$, $a \le b$, ce reprezintă intervale.

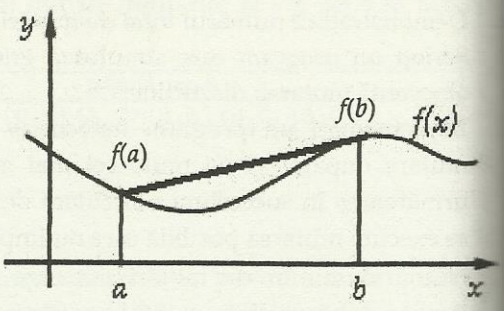
Date de ieșire. Pentru fiecare interval din fișierul de intrare, scrieți câte o linie în fișierul de ieșire cu valoarea integralei, ca în exemplu:



integrala.in	integrala.out
0 1 90.09 123.45 -45.6 23	I(0, 1) = 0.785383 I(90.09, 123.45) = 0.00299929 I(-45.6, 23) = 3.07615

Analiza problemei şi proiectarea soluţiei

Aria trapezului cu vârfurile de coordonate $(a, 0)$, $(b, 0)$, $(a, f(a))$ şi $(b, f(b))$ este: $(b-a)\left(\frac{f(a)+f(b)}{2}\right)$.



Cu ajutorul valorii *epsilon*, determinăm o condiţie de oprire pentru algoritmul recursiv, şi anume atunci când lungimea intervalului devine mai mică sau egală cu *epsilon*. În acest caz, integrala se va aproxima cu aria trapezului.

Program

```
#include <fstream>

const double eps=0.0001;

using namespace std;

double f(double x){
  return 1/(1+x*x);
}

double integral(double a, double b){
  if(b-a>eps){
    double c=(a+b)/2;
    return integral(a,c) + integral(c, b);
    } else{
      return (b-a)*(f(b)+f(a))/2;
    }
}

int main(){
  double a, b;
  ifstream in("integrala.in");
  ofstream out("integrala.out");
  while(in && !in.eof() && in>>a>>b){
    out << "I("<< a <<", " << b <<") = ";
    out << integral(a, b) << "  " << endl;
  }
  return 0;
}
```

Probleme propuse

1. Scrieți o variantă iterativă pentru problemă.
2. Modificați programul astfel încât să folosiți metoda dreptunghiului, care spune că zona dintre graficul funcției și axa Ox pe intervalul $[a, b]$ se poate aproxima cu aria dreptunghiului cu o latură $|f((a+b)/2)|$ și cealaltă lungimea intervalului.

Problema 4. *QuickSort* (sortarea rapidă)

În fișierul *QSort.in* se găsesc mai multe numere naturale. Implementați metoda *QuickSort* pentru a le sorta și scrieți rezultatul în fișierul *QSort.out*. Exemplu:

QSort.in	QSort.out
345 67 13 789 90 76 -45	-45 13 67 76 90 345 789

Analiza problemei și proiectarea soluției

QuickSort este un algoritm de sortare, ce a fost prezentat pentru prima dată de către C.A.R. Hoare (n. 1934) în 1962. *QuickSort* este în medie foarte rapid atunci când trebuie sortate mulțimi mari de date. Algoritmul alege un element aleator, din secvența de sortat, care reprezintă pivotul (punct de deviație). Apoi, secvența de date se împarte în două zone: una în care elementele sunt mai mici sau egale cu pivotul și una în care elementele sunt mai mari (în cadrul zonelor respective, elementele nu sunt sortate). Apoi, fiecare zonă se împarte recursiv în două zone, pe baza unui pivot corespunzător, până când se ajunge la zone cu un element (problemă elementară). *QuickSort* este, în medie, cel mai rapid algoritm de sortare.

În implementarea noastră, alegem ca pivot primul element din secvența $a[inf]$, $a[inf+1]$, ..., $a[sup]$ și folosim variabilele contor i și j, cu care ne deplasăm spre dreapta, respectiv spre stânga în secvența dată. După împărțirea datelor în două zone (prima cu elemente mai mici sau egale cu pivotul, a doua cu elemente mai mari decât pivotul), vom plasa elementul pivot între ele și vom apela recursiv metoda *quickSort()* pentru fiecare zonă. Condiția de terminare este ca dimensiunea secvenței să fie 1 ($inf=sup$).

Complexitatea. Dacă distribuția conduce în general la împărțirea în două zone, ale căror dimensiuni sunt aproape egale, atunci complexitatea algoritmului este $O(n \log n)$. Cel mai nefavorabil caz se produce atunci când cele două zone obținute au lungimi foarte diferite, complexitatea algoritmului devenind pătratică.

Program

```
#include <fstream>
#include <vector>

using namespace std;

void readData(vector<int> &a){
  int aux;
  ifstream f("QSort.in");
  while(f && !f.eof() && f>>aux)
    a.push_back(aux);
}

void quickSort(vector<int> &a, int inf, int sup){
  if(inf<sup){
    int pivot = a[inf], aux;
    int i = inf + 1, j = sup;
    while(i<=j){
      while(i<=sup && a[i] <= pivot) i++;
      while(j>=inf && a[j]  > pivot) j--;
      if(i<j && i<=sup && j>=inf){
        aux = a[i];
        a[i] = a[j];
        a[j] = aux;
        i++; j--;
      }
    }
    i--;
    a[inf] = a[i]; a[i] = pivot;
    quickSort(a, inf, i-1);
    quickSort(a, i+1, sup);
  }
}

void write(vector<int> a){
  ofstream f("QSort.out");
  for(int i=0; i<a.size(); i++) f << a[i] << " ";
}

int main(){
  vector<int> a;
  readData(a);
  quickSort(a, 0, a.size()-1);
  write(a);
  return 0;
}
```

Probleme propuse

1. Scrieți pe hârtie pașii algoritmului pentru exemplul dat (*QSort.in*).
2. Modificați programul de mai sus, astfel încât elementul pivot să fie ales întâmplător din intervalul dat.
3. Se consideră un vector cu n elemente. Scrieți un algoritm care returnează al k-lea element ($0 \le k < n$) al vectorului virtual sortat, fără a sorta însă vectorul dat.

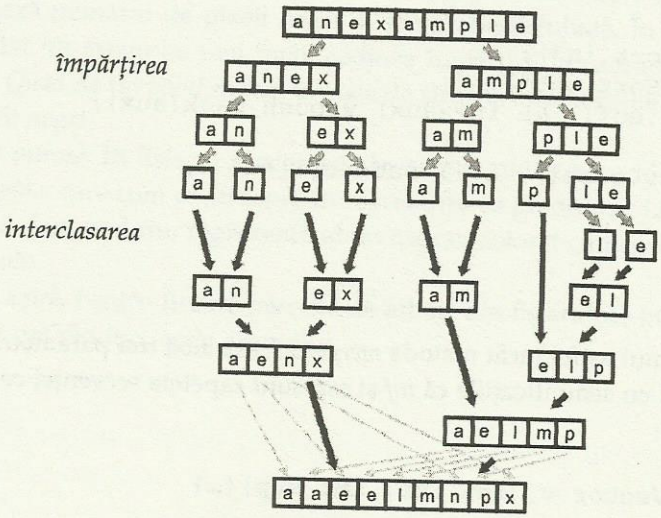
Problema 5. *MergeSort* (sortarea prin interclasare)

În fișierul *MSort.in* se găsesc mai multe numere naturale. Implementați metoda *MergeSort*, cu ajutorul căreia numerele date să fie ordonate crescător. Exemplu:

MSort.in	MSort.out
345 67 13 789 90 76 -45	-45 13 67 76 90 345 789

Analiza problemei și proiectarea soluției

MergeSort este, ca și *QuickSort*, o metodă eficientă de sortare. Ea procedează astfel: vectorul dat se împarte în două părți, care vor fi ordonate recursiv și apoi interclasate, pentru a obține vectorul inițial sortat. Complexitatea procesului de interclasare este liniară, pentru că ambele secvențe sunt deja sortate. Complexitatea algoritmului este $O(n \log n)$, iar, în practică, timpul de execuție este în medie mai mare decât cel obținut cu metoda *QuickSort*.



Program

```
#include <fstream>
#include <vector>

using namespace std;
template <class C>

void mergeSort(C& v){
  if (v.size()<=1) return;
  C v1(v.size()/2);
  C v2(v.size()-v1.size());
  for(int i=0; i<v1.size(); i=i+1) v1[i]=v[i];
  for(int i=0; i<v2.size(); i=i+1) v2[i]=v[i+v1.size()];
  mergeSort(v1);
  mergeSort(v2);
  int i1=0, i2=0;
  for(int k=0; k<v1.size()+v2.size(); k=k+1)
    if ((i2>=v2.size()) || (i1<v1.size() && v1[i1]<=v2[i2])){
      v[k] = v1[i1]; i1 = i1+1;
    }
    else{
      v[k] = v2[i2]; i2 = i2+1;
    }
}

int main(){
  vector<int> v;
  int aux;
  ifstream in("MSort.in");
  ofstream out("MSort.out");
  while(in && !in.eof() && in>>aux) v.push_back(aux);
  mergeSort(v);
  for(int i=0; i<v.size(); i++) out<<v[i]<<" ";
  return 0;
}
```

Probleme propuse

1. Modificaţi programul astfel încât metoda *mergeSort()* să aibă trei parametri: *Vector v, int inf* şi *int sup*, cu semnificaţiile că *inf* şi *sup* sunt capetele secvenţei ce trebuie ordonate în vector:

```
void mergeSort(Vector v, int inf, int sup){...}
```

2. Implementaţi o reprezentare grafică adecvată pentru algoritmul *MergeSort*.

Problema 6. Arbori *Quad* (*Quad-Trees*)

Arborii *Quad* sunt arbori ale căror noduri interioare au maxim patru succesori şi sunt utilizaţi, de exemplu, pentru prelucrarea fotografiilor. Arborii *Quad* de tip *point region* (*point region Quad-Trees*) sunt arbori *Quad* de un tip special, în care un nod are patru succesori nici unul. În această problemă, vom utiliza astfel de arbori *Quad*.

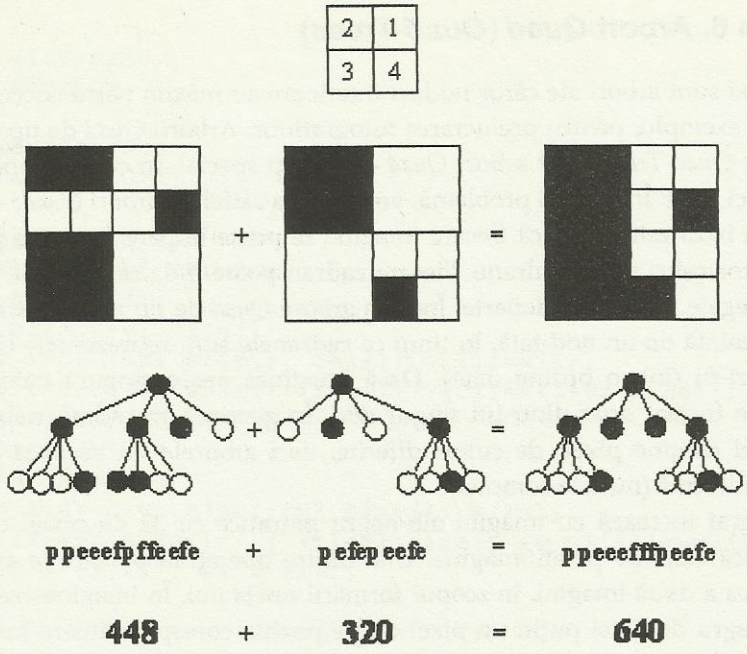
Ideea de bază este aceea că fiecare imagine se poate împărţi în patru zone egale, corespunzător celor patru cadrane. Fiecare cadran poate fi, la rândul său, împărţit în patru zone egale şi aşa mai departe. Într-un arbore *Quad* de tip *point region*, imaginea este reprezentată de un nod-tată, în timp ce cadranele sunt reprezentate ca fiind cele patru noduri-fii (într-o ordine dată). Dacă imaginea are o singură culoare, atunci arborele este format doar dintr-un singur nod. În general, un cadran trebuie divizat doar dacă el conţine pixeli de culori diferite, deci arborele nu va avea neapărat o adâncime uniformă (nu este complet).

Un fotograf lucrează cu imagini alb-negru pătratice cu 32 de pixeli orizontal şi vertical, adică 1024 de pixeli/imagine. Una dintre operaţiile pe care le execută este suprapunerea a două imagini, în scopul formării uneia noi. În imaginea rezultată, un pixel este negru dacă cel puţin un pixel de pe poziţia corespunzătoare în una dintre imagini este tot negru, altfel este alb (numai când ambii pixeli sunt albi). Fotograful crede că o fotografie va fi cu atât mai scumpă cu cât are mai mulţi pixeli negri. Aşa că, înainte de a suprapune două imagini, el trebuie să ştie câţi pixeli vor fi negri în imaginea rezultată, adică şi-ar dori să ştie câţi pixeli negri vor rezulta.

Scrieţi un program care, pe baza reprezentării ca arbori *Quad* a două fotografii, calculează numărul de pixeli negri în fotografia rezultată. În figura de mai jos, este prezentat un exemplu: mai întâi, ordinea împărţirii în cadrane, apoi imaginile, apoi arborii *Quad* de tip *point region*, urmate de şirurile în preordine şi, în final, de numărul de pixeli negri.

Date de intrare. În fişierul *quad.in* se găsesc mai multe perechi de arbori *Quad* de tip *point region*, care sunt consideraţi în reprezentarea preordine. Litera *p* simbolizează un nod-tată, *f* un nod-fiu, reprezentând un cadran colorat cu negru, iar *e* desemnează un cadran alb.

Date de ieşire. Pentru fiecare pereche de arbori din fişierul de intrare, scrieţi în fişierul *quad.out* trei rânduri, ca în exemplu:



quad.in	quad.out
ppeeefpffeefe pefepeefe	ppeeefpffeefe + pefepeefe = ppeeefffpeefe 448 + 320 = 640 ----------
peeef peefe	peeef + peefe = peeff 256 + 256 = 512 ----------
peeef peepefefe	peeef + peepefefe = peepefeff 256 + 128 = 384 ----------
ppeeefffpeefe peepefeff	ppeeefffpeefe + peepefeff = ppeeeffff 640 + 384 = 832 ----------
e pfffe	e + pfffe = pfffe 0 + 768 = 768 ----------

(ACM, West-European Regionals, 1994-1995, enunț modificat)

Analiza problemei şi proiectarea soluției

Metoda recursivă *unifyTrees()* calculează „suma" (numărul de pixeli negri) a doi arbori *Quad* reprezentați ca șiruri în preordine (*t1* şi *t2*) şi rezultatul este returnat în șirul *s*. Atunci când ambii arbori nu sunt frunze (noduri terminale), se vor extrage cei patru subarbori corespunzători cadranelor şi metoda este apelată recursiv pentru fiecare dintre cei patru (subarborele *i* al primului arbore este unificat cu subarborele *i* al celui

de-al doilea arbore, pentru $i = 1, ..., 4$). Pentru a extrage un subarbore, care începe la o poziție dată, folosim metoda:

```
void subTree(string& ss, string& sr, int &t){...}
```

unde şirul *ss* este reprezentarea arborelului dat, *sr* este reprezentarea subarborelui *Quad* care începe la poziția *t* în *ss*. Şirul dat se va parcurge caracter cu caracter de la stânga spre dreapta, începând cu poziția *t*. În variabila *n*, vom salva numărul de noduri care mai trebuie să fie citite. Dacă *n* devine zero, atunci ştim că am determinat un subarbore. Atunci când se citeşte un caracter *p*, ştim că mai trebuie să citim patru noduri succesoare; de aceea, *n* se incrementează cu 4. Deoarece *p* nu poate fi un nod-frunză, rezultă că *n* va fi micşorat cu 1. În total, se vor aduna în acest caz 3. Atunci când se citeşte o literă *f* sau *e* (frunză), atunci *n* se micşorează cu 1.

Program

```
#include <fstream>
#include <string>

using std::string;
using std::ifstream;
using std::ofstream;

bool isLeaf(char c){
  return
    ('e'==c || 'f'==c);
}


void subTree(string& ss, string& sr, int& t){
  sr.clear();
  if('p'==ss[t]){
    int n = 1;
    while (n && t<(int)ss.length()){
      if (isLeaf(ss[t])) n--;
      else  n += 3;
      sr  += ss[t++];
    }
  }
  else{
    sr=string(1, ss[t]);
    t++;
  }
}

int getValue(string ss, int v){
```

```
  if ('p'==ss[0]){
    string sr;
    int t = 0;
    int suma = 0;
    t = 1;
    for(int i=0; i<4; i++){
      subTree(ss, sr, t);
      suma += getValue(sr, v/4);
    }
    return suma;
  }
  else
    if ('f'== ss[0])
      return v;
  return 0;
}

void unifyTrees(string t1, string t2, string& s){
  if ('p'==t1[0] && 'p'==t2[0]) {
    s += "p";
    int i1 = 1, i2 = 1;
    string sr1, sr2;
    for(int i=0; i<4; i++){
      subTree(t1, sr1, i1);
      subTree(t2, sr2, i2);
      unifyTrees(sr1, sr2, s);
    };
  }
  else
    if (t1[0]=='f' || t2[0]=='f') s += "f";
    else if ('e'==t2[0]) s += t1;
    else if ('e'==t1[0]) s += t2;
}

void write(ofstream& out, string t1, string t2, string s){
  out << t1 <<" + " << t2 << " = " << s << std::endl;
  out << getValue(t1, 1024)<<" + "
      << getValue(t2, 1024) << " = "
      << getValue(s, 1024) << std::endl;
  out << "-----------------------------------\n";
}

int main(){
  string t1, t2, s;
  ifstream in("quad.in");
  ofstream out("quad.out");
  while(in && !in.eof() && in>>t1>>t2){
    s.clear();
```

```
    unifyTrees(t1, t2, s);
    write(out, t1, t2, s);
  }
  return 0;
}
```

Probleme propuse

1. Lungimea laturii unei astfel de fotografii pătratice este n. Calculați aria ce este acoperită de pixeli negri pentru un arbore *Quad* dat.
2. Scrieți un program pentru o reprezentare grafică pe cât posibil mai convenabilă a celor doi arbori *Quad* dați şi a arborelui rezultat.
3. *Reduceri de termeni.* Un termen este definit inductiv astfel:
   a) X este termen;
   b) dacă A şi B sunt termeni, atunci $(A\ B)$ este termen;
   c) un termen se obține doar aplicând regulile de mai sus.

   Un termen de forma $(((X\ A)\ B)\ C)$, unde A, B şi C sunt termeni, se poate înlocui cu $((A\ C)(B\ C))$. Reducerea unui termen înseamnă aplicarea acestei reguli. Spunem că un termen este redus, dacă nu conține nici un termen reductibil. Un termen se consideră că este subtermen propriu. Scrieți un program care să citească termenii şi să scrie termeni corespunzători reduşi.

   Date de intrare. Fişierul *termeni.in* conține mai multe linii cu termeni, câte unul pe linie. Fiecare linie va conține cel mult 80 de caractere. Termenii se consideră corecți sintactic. Liniile de intrare se vor sfârşi cu o linie care conține caracterul 0.

   Date de ieşire. Pentru fiecare linie din fişierul de intrare se vor scrie în fişierul *termeni.out* câte o linie reprezentând termenul respectiv redus, apoi o linie reprezentând cifre în pozițiile în care termenul de ieşire are paranteze, urmată de o linie vidă. Un termen fără paranteze va fi urmat de două linii vide. Aceste cifre marchează nivelul de imbricare al parantezelor în respectivele poziții. Acest nivel este ≤ 9, iar numărul de reduceri este finit. Caracterele posibile în termenii de ieşire sunt X, (şi). Exemplu:

termeni.in	termeni.out
(((XX)X)X) (((XX)(XX))X) (XX) 0	((XX)(XX)) 01 11 10 ((XX)((XX)X)) 01 112 2 10 (XX) 0 0

(ACM, Est-European Regionals, 1996)

Problema 7. Transformata *Fourier* discretă (*DFT*)

Transformata *Fourier* discretă este cel mai important instrument în prelucrarea semnalelor şi este folosită pentru codificarea şi decodificarea semnalelor la nivelul frecvenţelor. Este folosită, de exemplu, de către cunoscutul algoritm de compresie *MP3*. Scrieţi un program care calculează transformata *Fourier* discretă pentru vectorul de coeficienţi $a=(a_0,a_1,...,a_{n-1})$.

Fie $\omega_n^0,\omega_n^1,\omega_n^2,...,\omega_n^{n-1}$ cele n rădăcini ale ecuaţiei $X^n-1=0$ (rădăcinile unităţii).

Definim polinomul $P(x)=\sum_{j=0}^{n-1} a_j \cdot x^j$. Transformata *Fourier* discretă a vectorului a este atunci $y=(P(\omega_n^0),P(\omega_n^1),...,P(\omega_n^{n-1}))$ şi scriem $y = DFT_n(a)$.

Date de intrare. În fişierul *fourier.in* se găsesc coeficienţii vectorului a, ale cărui elemente sunt numere complexe (date prin părţile reală şi imaginară). Numărul elementelor lui a este o putere a lui 2.
Date de ieşire. Scrieţi în fişierul *fourier.out* transformata *Fourier* discretă y a vectorului de coeficienţi a. Exemplu:

fourier.in	fourier.out
3.45 2.1 0.8 7.6 -5.4 3.1 -7.4 2.1 7.6 43.21 6.54 32.1 8.76 43 3.12 -3.21	17.47 130 52.6993 -87.8464 -33.12 10.83 -33.8606 -17.3687 11.35 52.82 18.8007 -22.6936 48.5 -12.41 -54.2394 -36.5313

Analiza problemei şi proiectarea soluţiei

Metoda naivă de rezolvare are complexitatea $O(n^2)$. Dacă n este o putere a lui 2, atunci metoda poate fi îmbunătăţită: transformata *Fourier* rapidă (engl., *Fast Fourier Transformation, FFT*) are complexitatea $O(n \log n)$.

FFT foloseşte principiul *Divide et Impera*. Metoda utilizează două noi polinoame $A_0(x)$ şi $A_1(x)$, ambele de grad $n/2$, care se formează din coeficienţii polinomului $P(x)$, separându-i pe cei corespunzători puterilor pare, respectiv impare:

$$A_0(x)=a_0+a_2x+a_4x^2+...+a_{n-2}x^{\frac{n}{2}-1}$$

$$A_1(x)=a_1+a_3x+a_5x^2+...+a_{n-1}x^{\frac{n}{2}-1}$$

Rezultă: $P(x) = A_0(x^2) + xA_1(x^2)$ (1)

Valorile lui $P(x)$ în $\omega_n^0, \omega_n^1, \omega_n^2, ..., \omega_n^{n-1}$ vor fi calculate astfel:

1. Calculăm $A_0(x)$ şi $A_1(x)$ (ambele de grad $n/2$) pentru valorile $(\omega_n^0)^2, (\omega_n^1)^2, (\omega_n^2)^2, ..., (\omega_n^{n-1})^2$.
2. Cu ajutorul formulei (1) obţinem $P(x)$.

Algoritmul în pseudocod:

```
ALGORITM_FFT(Vector a)
    1.  n ← size(a)
    2.  If (n = 1) return a
    3.  ω_n ← cos(2π/n) + i sin(2π/n)
    4.  ω ← 1
    5.  A_0 ← (a_0, a_2, ..., a_{n-2})
    6.  A_1 ← (a_1, a_3, ..., a_{n-1})
    7.  Y_0 ← Divide_Et_ImperaFFT(A_0)
    8.  Y_1 ← Divide_Et_ImperaFFT(A_1)
    9.  For (k ← 0; k ≤ n/2 − 1; step 1)
            y_k ← Y_0(k) + ω · Y_1(k)
            y_{k+n/2} ← Y_0(k) − ω · Y_1(k)
            ω ← ω · ω_n
        End_For
   10.  return y
END_ALGORITM _FFT(Vector a)
```

Vom scrie clasa *Complex* pentru a reprezenta numerele complexe, unde vom supraîncărca operatorii (+, –, *, >>, <<). Metoda *FFT()* furnizează transformata *Fourier* pe baza algoritmului de mai sus. Mai avem nevoie de constanta π, care poate fi calculată cu una dintre formulele: $\tan\frac{\pi}{4} = 1$ sau $\cos\frac{\pi}{2} = 0 \Rightarrow \pi = 4 \cdot \arctan 1$ sau $\pi = 2 \cdot \arccos 0$.

Program

```
#include <fstream>
#include <vector>
#include <cmath>

double const pi(4*atan(1.0));
using namespace std;
class Complex{
  private:
    double _re, _im;
  public:
    Complex (){_re=0; _im=0;};
    Complex(double re, double im){
      _re=re; _im=im;
    }
    ~Complex(){};
    Complex operator+(Complex&);
    Complex operator-(Complex&);
    Complex operator*(Complex&);
    friend istream& operator>>(istream&, Complex&);
    friend ostream& operator<<(ostream&, Complex&);
    void setRe(double re){ _re=re;};
    void setIm(double im){ _im=im;};
};

inline Complex Complex::operator+(Complex&z){
  return Complex(_re+z._re, _im+z._im);
}

inline Complex Complex::operator-(Complex&z){
  return Complex(_re-z._re, _im-z._im);
}

inline Complex Complex::operator*(Complex&z){
  return Complex(_re*z._re-_im*z._im, _re*z._im+_im*z._re);
}

istream& operator>>(istream& is, Complex& z){
  is >>z._re>>z._im;
  return is;
}

ostream& operator<<(ostream& os, Complex& z){
  os<<z._re<<"  "<<z._im;
  return os;
}
```

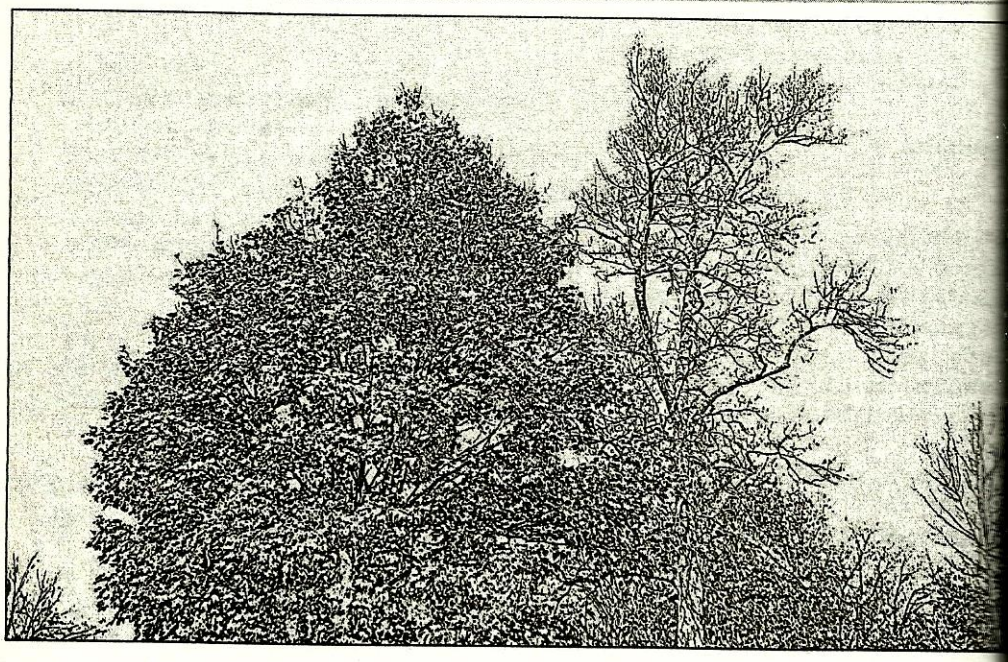
```
vector<Complex> FFT(vector<Complex> a){
  int i, n=a.size();
  if(n<=1) return a;
  Complex wn(cos(2*pi/n), sin(2*pi/n));
  Complex w(1.0, 0.0);
  vector<Complex> a0, a1, y0, y1, y;
  for(i=0; i<n; i++){
    if(i%2==0)a0.push_back(a[i]);
    else a1.push_back(a[i]);
    y.push_back(Complex(0, 0));
  }
  y0=FFT(a0);
  y1=FFT(a1);
  for(i=0; i<n/2; i++){
    y[i]=y0[i]+w*y1[i];
    y[i+n/2]=y0[i]-w*y1[i];
    w=w*wn;
  }
  return y;
}

int main(){
  vector<Complex> a, y;
  Complex w, aux, E, res;
  ifstream in("fourier.in");
  ofstream out("fourier.out");
  while(in && !in.eof() && in>>aux){
    a.push_back(aux);
  }
  int n=a.size();
  y=FFT(a);
  for(int i=0; i<y.size(); i++)
      out <<y[i]<<endl;
  return 0;
}
```

Probleme propuse

1. Dezvoltați clasa *Complex* cu operatorii +=, *=, -= și folosiți-i apoi în metoda *FFT()*.
2. Implementați un algoritm naiv de calcul al transformatei *Fourier* discrete pentru orice grad al polinomului coeficienților și folosiți-o în programul de mai sus. Analizați următoarea variantă:

```
vector<Complex> y;
w.setRe(1);
w.setIm(0);
```

```
for(int k=0; k<n; k++){
   Complex yk(0, 0);
   E.setRe(cos(2*pi*k/n)); E.setIm(sin(2*pi*k/n));
   w.setRe(1); w.setIm(0);
   for(int j=0; j<n; j++){
     yk = yk+a[j]*w;
     w=w*E;
   }
   y.push_back(yk);
}
```



Arbori – toamnă în Ottawa

Capitolul 10
Backtracking

14 probleme complet rezolvate, 64 de probleme propuse

- Problema celor *n* regine
- Observații generale cu privire la metoda *Backtracking*
- Problema celor *n* ture pe tabla de șah
- Problema turelor pe primele *m* linii ale unei table de șah
- Problema turelor în scară pe primele *m* linii ale unei table de șah
- Tabăra prieteniei
- Partițiile unui număr natural
- Referate la geografie
- Toate drumurile calului pe tabla de șah
- Problema fotografiei
- Rostogolirea mingii
- Portocalosport
- Sudoku
- Căsuța lui Moș Crăciun
- Compactarea șabloanelor de test
- Încă 12 probleme propuse

Nu merge unde te duce drumul,
ci mergi unde nu există nici un drum şi lasă o urmă.
Radolph Waldo Emerson

Metoda *Backtracking* este foarte cunoscută şi cu ajutorul ei se pot rezolva foarte multe probleme. Una dintre acestea, ce apare în aproape orice curs despre *Backtracking*, este „Problema celor *n* regine". Cineva chiar afirma cândva: „Când se spune *Backtracking*, se spune, de fapt, «*n* regine» şi când se spune «*n* regine» se spune, de fapt, *Backtracking*!". De aceea, vom începe capitolul cu această problemă, simplu de descris, dar nu chiar aşa de simplu de rezolvat.

Problema 1. Problema celor *n* regine

Istoric. Problema celor opt regine a fost prezentată pentru prima dată în 1848 de către maestrul bavarez de şah Max Bezzel (1824-1871), în *Berliner Zeitung*, unde acesta se întreba care este numărul de soluţii. Acest număr este 92 şi a fost introdus de către dr. Franz Nauck în 1850, în ziarul *Leipziger Illustrierten Zeitung*. Pentru că şi Carl Friederich Gauß s-a ocupat de problemă, se indică deseori – fals – că el ar fi răspuns la întrebare. Abia în 1991, B. Bernhardsson prezenta în *ACM SIGART Bulletin*, volumul 2, numărul 7, o soluţie pentru problema generalizată. În 1992, Demirörs, Rafraf şi Tanik au introdus o echivalenţă între pătratele magice şi problema celor *n* regine.
Problemă. Fie o tablă de şah pătratică de dimensiune $n \times n$ ($4 \leq n \leq 20$). Determinaţi toate posibilităţile de a amplasa *n* regine pe această tablă, astfel încât să nu existe regine care se atacă reciproc. Două regine se atacă reciproc dacă ele se află pe aceeaşi linie, coloană sau diagonală.
Date de intrare. De la tastatură se introduce numărul de linii *n* al tablei de şah.
Date de ieşire. Scrieţi în fişierul de ieşire *nQueens.out* numărul coloanei pentru fiecare regină de pe liniile 1, 2, ..., *n* respectiv, ca în exemplul de mai jos. Pe ultima linie afişaţi *n* şi numărul soluţiilor. Exemplu:

Tastatură	nRegine.out
n = 4	2 4 1 3 3 1 4 2 4 2
n = 8	1 5 8 6 3 7 2 4 1 6 8 3 7 4 2 5 ... 6 3 1 7 5 8 2 4 ... 8 3 1 6 2 5 7 4 8 4 1 3 6 2 7 5 8 92

Analiza problemei şi proiectarea soluției

Presupunem că $n = 4$ şi că vrem să determinăm toate posibilitățile de a amplasa patru regine pe această tablă, astfel încât oricare două dintre ele să nu se atace reciproc.

Cum putem să procedăm? Mai întâi amplasăm o regină în câmpul (1, 1), după care încercăm să poziționăm o regină pe linia a doua. Observăm că primul câmp posibil este (2, 3), deoarece primele două poziții sunt în conflict cu regina de pe prima linie. Deci a doua regină se amplasează pe poziția (2, 3); vezi figura 10.1.

Apoi, observăm că nu este posibil să poziționăm o regină pe linia a treia: primul câmp este atacat de către regina de pe prima linie, iar următoarele trei câmpuri nu sunt accesibile din cauza reginei de pe a doua linie (figura 10.2). Cu această amplasare a primelor două regine nu are sens să continuăm căutarea unei soluții: este clar că nu există nici una!

Ne întoarcem la a doua linie şi deplasăm regina cu o poziție la dreapta (figura 10.3). Pe linia a treia putem aşeza o regină pe a doua poziție (figura 10.4). Ultima regină nu poate fi plasată pe al patrulea rând, deoarece se atacă cu reginele deja amplasate (figura 10.5).

Am învățat deja că în acest caz trebuie să ne deplasăm la rândul anterior şi să mutăm regina de acolo cu o poziție la dreapta. Remarcăm că nici pe linia a treia nu avem succes, deoarece câmpurile 3 şi 4 se atacă cu regina de pe linia a doua. Deci ne deplasăm pe linia a doua şi observăm că aici ne aflăm pe ultimul câmp.

Trebuie aşadar să ne întoarcem la prima linie şi să deplasăm regina de aici pe poziția a doua (figura 10.6). Pentru a doua regină este valabil numai ultimul câmp (figura 10.7). A treia regină se poate aşeza doar pe primul câmp (figura 10.8). În sfârşit, putem poziționa ultima regină pe al treilea câmp (figura 10.9). În acest moment, cele patru regine sunt amplasate şi am determinat deci o soluție a problemei.

Pentru a mai găsi o soluție posibilă, vom urma metoda deja descrisă. După câțiva paşi ajungem la soluția din figura 10.10 (se observă că aceasta este o oglindire a primeia).

Căutarea a încă unei soluții ne conduce la figura 10.11, caz în care trecerea la rândul anterior (ar fi 0!) nu mai este posibilă. În acest moment, algoritmul se încheie şi toate soluțiile pentru $n = 4$ au fost determinate.

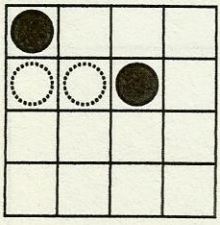


Figura 10.1

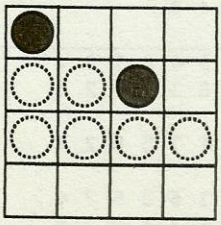


Figura 10.2

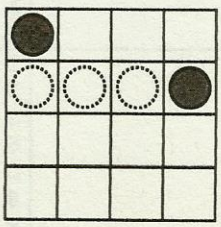


Figura 10.3

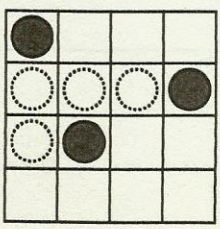


Figura 10.4

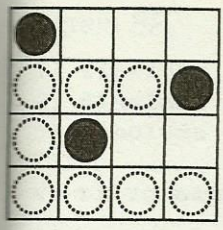


Figura 10.5

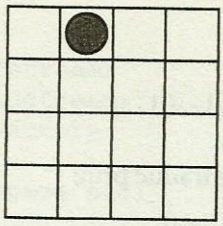


Figura 10.6

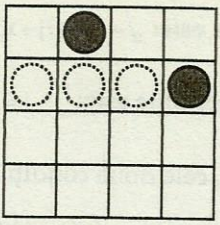


Figura 10.7

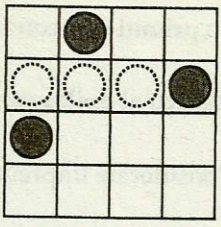


Figura 10.8

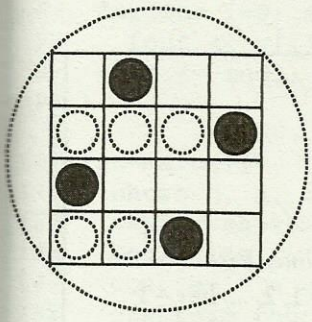


Figura 10.9

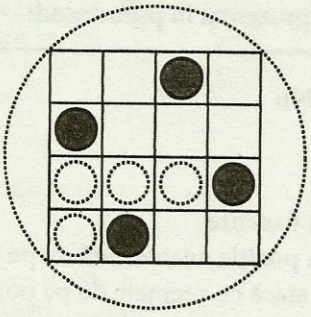


Figura 10.10

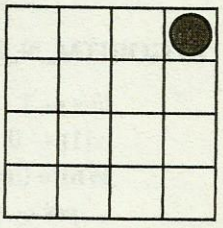
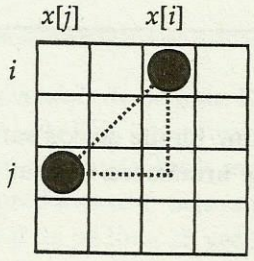
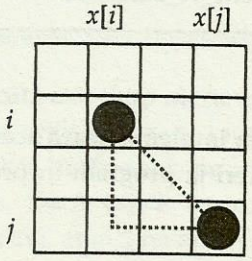


Figura 10.11

Observații:

1. Pe fiecare rând se află exact o regină.
2. Pe fiecare coloană se află exact o regină.
3. Putem reprezenta soluția ca un tablou unidimensional $x[]$ cu semnificația: $x[i]$ este coloana pe care se plasează a i-a regină. Prima soluție se poate reprezenta ca $x[4] = \{2, 4, 1, 3\}$, a doua soluție ca $x[4] = \{3, 1, 4, 2\}$. Această reprezentare ne asigură că pe fiecare rând se află numai o regină.
4. O condiție necesară ca reginele i şi j să nu se atace este: $x[i] \neq x[j]$ (coloanele trebuie să fie diferite).
5. Pentru oricare două regine i şi j, $i < j$, există încă o condiție necesară: nu este permis ca ele să se afle pe aceeaşi diagonală. Putem avea una dintre următoarele posibilități:





- în primul caz, condiția este: $j-i=x[i]-x[j]$;
- pentru al doilea caz, condiția este: $j-i=x[j]-x[i]$.

Considerate împreună, cele două condiții se traduc prin:

$$j-i=|x[i]-x[j]|.$$

Algoritmul poate fi astfel prezentat în pseudocod:

```
ALGORITM_N_REGINE
        line ← 1
        x[1] ← 0
        While (line > 0) Execute
          pos ← prima poziție neanalizată de pe linia line, care
                nu se atacă cu reginele de pe pozițiile 1, 2, ..., line – 1
          If (pos există) Then
              x[line] ← pos
            If (line = n) Then
               WriteSolution(x[1], x[2], ..., x[n])
            End_If
            Else
             If (line < n) Then
               line ← line + 1    // pas înainte (forward)
               x[line] ← 0        // setăm coloana de start cu 0
             End_If
             Else Execute
               line ← line – 1    // pas înapoi (backward)
             End_Else
          End_If
        End_While
END_ALGORITM_N_REGINE
```

Următorul program implementează acest algoritm. Liniile de cod sunt numerotate, pentru a ne putea referi la program în problemele următoare. Notăm programul cu *BP*.

Program *BP*

```
 1: #include <fstream>
 2: #include <iostream>
 3: #include <vector>

 4: using namespace std;

 5: void writeSolution(vector<short> &x, ofstream &fout){
 6:  for(int i=0; i<x.size(); i++)
 7:    fout << x[i]+1 << " ";
 8:  fout << endl;
 9: }

10: int main(){
11:   short n, k, i, noSol=0;
12:   vector<short> x;
13:   bool flag;
14:   cout << " n = "; cin >> n;
15:   ofstream fout("nRegine.out");
16:   x.push_back(-1);
17:   while(!x.empty()){
18:     k = x.size()-1;
19:     flag = false;
20:     while(!flag && x[k]<n-1){
21:       x[k]++; flag = true;
22:       for(i=0; i<k; i++)
23:         if(x[i]==x[k]||abs(x[i]-x[k])==k-i) flag=false;
24:     }
25:     if(flag){
26:       if(k==n-1){
27:         writeSolution(x, fout);
28:         noSol++;
29:       }else
30:         x.push_back(-1);
31:     }else x.pop_back();
32:   }
33:   fout << n << " " << noSol;
34:   return 0;
35: }
```

Observații. Bucla *while* de pe linia 17 se execută atât timp cât ne aflăm pe un rând valid (între 0 şi $n-1$). Pe linia 20 urmează o buclă *while*, în care se caută următoarea coloană posibilă pentru $x[k]$, astfel încât regina plasată în rândul k, pe coloana $x[k]$ nu se atacă cu reginele predecesoare deja amplasate. Dacă există o poziție „neatacată" $x[k]$ (instrucțiunea *if* de pe linia 25 verifică aceasta), atunci ne aflăm pe ultimul rând (linia 26) – deci am găsit o soluție – sau ne aflăm pe un rând în interior şi trebuie să coborâm

pe următorul (linia 30 setează coloana cu 0). Dacă pe rândul k nu există nici un câmp „neatacat", atunci ne întoarcem cu un rând înapoi (linia 31).

La rularea programului, remarcăm că timpul de execuție creşte foarte repede odată cu n, deoarece complexitatea algoritmului este exponențială.

Observații generale cu privire la metoda *Backtracking*

Chiar dacă *Backtracking* îmbunătăţeşte căutarea exhaustivă (de exemplu, nu generăm vectorul (1 1...1) şi verificăm apoi condițiile ca acesta să fie soluție, ci decidem mult mai devreme că (1 1...) nu poate conduce la o soluție!), timpul de execuție creşte exponențial. De aceea, metoda *Backtracking* ajunge să fie folosită doar atunci când s-a verificat că nu există o metodă mai eficientă de rezolvare a problemei (de exemplu, un algoritm polinomial). Pentru clasa problemelor care se rezolvă cu această metodă sunt valabile:

- de obicei, pentru un set de date de intrare se cer toate soluțiile posibile;
- o soluție se poate reprezenta ca un vector, care nu este necesar să aibă lungime fixă;
- o soluție poate fi construită pas cu pas: $x[1]$, $x[2]$, ..., $x[k]$;
- în momentul k se poate testa dacă valoarea $x[k]$ pentru al k-lea element poate conduce la o soluție, atunci când valorile $x[1]$, $x[2]$, ..., $x[k–1]$ au fost deja fixate;
- putem să decidem dacă am găsit o soluție a problemei.

În funcție de tipul soluției, o problemă rezolvabilă folosind *Backtracking* poate fi clasificată astfel: *Backtracking liniar cu lungime fixă* (problema celor n regine), *Backtracking cu lungime variabilă* (generarea partițiilor unui număr natural), *Backtracking în plan* (toate drumurile calului pe tabla de şah, problema labirintului, problema fotografiei, problema mingii).

Considerăm mai departe algoritmul general pentru *Backtracking*. Presupunem că elementele vectorului $x[]$ pot să ia valori din mulțimile A_1, A_2, ..., respectiv A_n. Algoritmul se poate descrie astfel:

```
ALGORITM_Backtracking_Iterativ
k ← 1
While (k ≥ 1) Do
  If ( ∃(candidat încă netestat din Ak pentru x[k]) ) Then
    x[k] ← candidat încă netestat din Ak
           pentru al k-lea element
    If (x[1], x[2], ..., x[k] este soluție) Then
       procesează(x[1], x[2], ..., x[k])
    Else k ← k + 1                          // pas înainte
  Else k ← k – 1                            // pas înapoi
End_While
END_ALGORITM_Backtracking_Iterativ
```

Metoda *Backtracking* se poate implementa şi recursiv, de exemplu, cu un algoritm:

ALGORITM_Backtracking_Recursiv(*k*)
  For (toate valorile posibile pentru *x*[*k*]) **Execute**
    If (*x*[1], *x*[2], ..., *x*[*k*] poate conduce la o soluţie) **Then**
      If (*x*[1], *x*[2], ..., *x*[*k*] este soluţie) **Then**
        procesează (*x*[1], *x*[2], ..., *x*[*k*])
      Else *Backtracking_Recursiv*(*k* + 1)
  End_For
END_ALGORITM_Backtracking_Recursiv(*k*)

Când rezolvăm o problemă folosind *Backtracking recursiv*, executăm apelul *back*(1).

Probleme propuse

1. Numiţi tipurile de soluţie pentru problemele rezolvabile folosind metoda *Backtracking*.
2. Modificaţi programul astfel încât să folosiţi tipul *std::stack<short>* şi operaţiile specifice pentru stivă *top*, *pop* şi *push*, în locul vectorului *vector<short>*. Unde apare problema?
3. Scrieţi un program ce rezolvă problema recursiv. De exemplu:

```
void back(vector<short> x, short n, ofstream &fout){
  short i, xk, k=x.size();
  bool flag;
  for(xk=0; xk<n; xk++){
    flag = true;
    if(x.size()==k+1) x.pop_back();
    for(i=0; flag && i<k; i++)
      if(x[i]==xk || abs(xk-x[i])==k-i) flag=false;
    if(flag){
      x.push_back(xk);
      if(k==n-1) writeSolution(x, fout);
      else back(x, n, fout);
    }
  }
}
```

Pentru a înţelege mai bine metoda, notaţi-vă paşii şi valorile intermediare pe o foaie de hârtie. De ce se foloseşte instrucţiunea: *if(x.size() == k + 1) x.pop_back();* ?

4. *Pătrate magice.* Un pătrat magic este o matrice pătratică cu numărul de linii şi de coloane *n*, care conţine numerele naturale 1, 2, ..., n^2 şi în care sumele elementelor

de pe aceeași linie, coloană și diagonală este identică. De exemplu, un pătrat magic de dimensiune 4:

$$A = \begin{pmatrix} 16 & 3 & 2 & 13 \\ 5 & 10 & 11 & 8 \\ 9 & 6 & 7 & 12 \\ 4 & 15 & 14 & 1 \end{pmatrix}$$

Demonstrați că pentru un număr natural n dat, suma elementelor de pe fiecare linie, coloană și diagonală este $\frac{n(n^2+1)}{2}$. Scrieți un program care listează în fișierul *patrMagice.out* toate pătratele magice de dimensiune 4. Scrieți și o metodă care să verifice dacă o matrice dată de dimensiune $n \times n$, $4 \le n \le 50$ este pătrat magic.

Problema 2. Problema celor *n* ture pe tabla de șah

Fie o tablă de șah de dimensiune $n \times n$. Determinați toate posibilitățile de a amplasa n ture pe tabla dată, astfel încât să nu existe printre ele perechi care se atacă (două ture se atacă reciproc atunci când se află pe aceeași linie sau pe aceeași coloană).
Date de intrare. De la tastaură, se introduce numărul natural n ($4 \le n \le 20$).
Date de ieșire. Scrieți în fișierul *nTure.out* pozițiile coloanelor pentru fiecare tură, ca în exemplul de mai jos, urmate de n și numărul soluțiilor pe ultima linie a fișierului.
Exemplu:

Tastatură	nTure.out
n = 5	1 2 3 4 5 1 2 3 5 4 ... 5 4 3 1 2 5 4 3 2 1 5 120

Analiza problemei și proiectarea soluției

Problema este foarte asemănătoare cu problema reginelor. Condiția este ca oricare două ture să nu se atace reciproc, adică să nu se găsească pe aceeași linie sau coloană. Asemănarea ne conduce la faptul că o soluție a problemei celor n regine este soluție și pentru problema turelor. Vom reprezenta soluția tot cu ajutorul unui vector $x[]$, cu aceeași semnificație. Pentru că dispare o condiție (aceeași diagonală pentru două ture este permisă), numărul soluțiilor crește. Singura schimbare apărută în programul anterior este la pasul în care se testează dacă $x[k]$ este candidat la o soluție parțială, atunci când valorile $x[0]$, $x[1]$, ..., $x[k-1]$ sunt fixate: nu mai este necesară verificarea diagonalelor. În afara liniei 15 din programul anterior, unde se introduce un alt nume pentru fișierul de ieșire, se va mai modifica și linia 23 astfel:

```
23:          if(x[i]==x[k]) flag=false;
```

(dacă tura i se află pe aceeaşi coloană cu tura k, înseamnă că vectorul parţial $x[1]$, $x[2]$, ..., $x[k]$ nu este un candidat pentru o soluţie!).

Observaţie. Aceasta este, de fapt, generarea tuturor permutărilor, deci numărul soluţiilor pentru un n dat este $n!$.

Probleme propuse

1. Scrieţi o implementare recursivă pentru problemă.
2. Permutările sunt listate lexicografic. Modificaţi programul astfel încât ele să fie listate colexicografic.

Problema 3. Problema turelor pe primele *m* linii ale unei table de şah

Găsiţi toate posibilităţile de a amplasa un număr maxim de ture pe primele m linii ale unei table de şah pătratice de dimensiune n, astfel încât oricare două dintre ele să nu se atace reciproc ($4 \le n \le 20$, $1 \le m \le n$).
Date de intrare. Valorile n şi m se citesc de la tastatură.
Date de ieşire. Toate soluţiile problemei se vor scrie în fişierul *nmTure.out*, iar pe ultima linie se vor scrie n, m şi numărul de soluţii, ca în exemplu:

Tastatură	nmTure.out
n = 6 m = 4	1 2 3 4 1 2 3 5 ... 6 5 4 2 6 5 4 3 6 4 360

Analiza problemei şi proiectarea soluţiei

Numărul maxim de ture cu condiţiile date este m. Schimbarea faţă de problema anterioară este că se vor genera doar primele m elemente ale vectorului. Modificăm programul *BP* cu citirea şi scrierea în plus a parametrului m, cu specificarea unui alt fişier de ieşire şi cu următoarele două linii:

```
23: if(x[i]==x[k]) flag=false; // (nu se vor testa diagonalele)

26: if(k==m-1){ // (o soluţie a fost găsită atunci când m elemente sunt
                //determinate)
```

Observație:
Aceasta este problema generării aranjamentelor de n luate câte m. Numărul soluțiilor este aşadar: $\frac{n!}{(n-m)!} = n(n-1)(n-2)...(n-m+1)$.

Probleme propuse

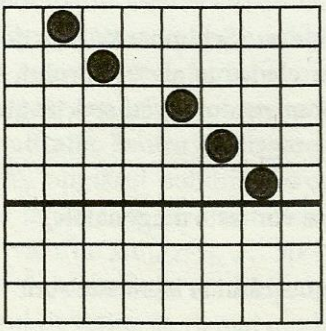
1. Scrieți un program recursiv ce rezolvă problema.
2. Aranjamentele sunt generate lexicografic. Modificați programul astfel încât acestea să fie generate colexicografic.

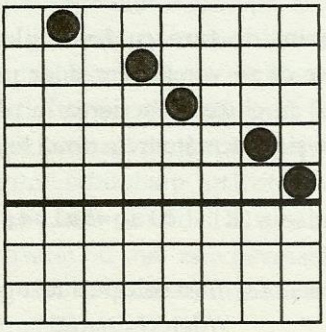
Problema 4. Problema turelor în scară pe primele *m* linii ale unei table de şah

Găsiți toate posibilitățile de a amplasa crescător un număr maxim de ture pe primele m linii ale unei table de şah de dimensiune $n \times n$, astfel încât oricare două dintre ele să nu se atace reciproc ($4 \le n \le 20$, $1 \le m \le n$).
Date de intrare. Numerele naturale n şi m se vor introduce de la tastatură.
Date de ieşire. Toate soluțiile se vor scrie în fişierul *nmCTure.out*, pe ultima linie scrieți n, m şi numărul soluțiilor, ca în exemplu:

Tastatură	nmCTure.out
n = 6 m = 4	1 2 3 4 1 2 3 5 ... 2 4 5 6 3 4 5 6 6 4 15

Definiție:
Spunem că secvența de ture $x[1]$, $x[2]$, ..., $x[k]$ este *crescătoare* dacă pentru orice $i, j \in \{1, 2, ..., k\}$, $i < j$, are loc: $x[i] < x[j]$. De exemplu, pentru $n = 8$ şi $m = 5$ sunt două soluții posibile:





Analiza problemei şi proiectarea soluției

Diferența față de problema anterioară este că trebuie să amplasăm crescător cele m elemente. O posibilitate ar fi ca în linia 30 a programului *BP* să inițializăm $x[k]$ cu $x[k-1]$. Operați în program această modificare.

Observație:
Aceasta este problema combinărilor de n luate câte m. Numărul soluțiilor este aşadar: $\frac{n!}{m!(n-m)!} = \frac{n(n-1)(n-2)...(n-m+1)}{m!}$.

Probleme propuse

1. Scrieți o implementare recursivă a programului.
2. Transformați modalitatea lexicografică de listare a combinărilor în cea colexicografică.

Problema 5. Tabăra prieteniei

În tabăra prieteniei se organizează în fiecare seară un foc de tabără. În prima seară, toți participanții se aşază în jurul focului. În a doua seară aceştia trebuie să se aşeze astfel încât cei care au fost vecini în prima seară să nu mai stea alături. Determinați toate posibilitățile de a-i aşeza pe cei n participanți în a doua seară în jurul focului. Presupunem că, în prima seară, aceştia sunt numerotați cu 1, 2, ..., n.
Date de intrare. Numărul natural n se introduce de la tastatură ($4 \le n \le 20$).
Date de ieşire. Scrieți toate posibilitățile de aşezare în fişierul *prieteni.out*. Ultima linie va conține n şi numărul de posibilități, ca în exemplu:

Tastatură	prieteni.out
n=5	1 3 5 2 4 ... 3 5 2 4 1 ... 5 3 1 4 2 5 10

Analiza problemei şi proiectarea soluției

În afara condiției că toate numerele din vectorul $x[]$ trebuie să fie diferite două câte două, trebuie să mai verificăm că diferența a două elemente vecine nu este egală cu 1 în modul, dar nici cu $n-1$ (n şi 1 au fost vecini!). În programul *BP*, liniile 22 şi 23 se vor înlocui cu:

```
if(k==n-1&&(abs(x[k]-x[0])==1||abs(x[k]-x[0])==n-1))
                                                    flag=false;
if(k>0&&(abs(x[k]-x[k-1])==1||abs(x[k]-x[k-1])==n-1))
                                                    flag=false;
for(i=0; i<k; i++) if(x[i]==x[k]) flag=false;
```

Probleme propuse

1. Rezolvați recursiv problema.
2. Găsiți formula de calcul a numărului de soluții în funcție de n.

Problema 6. Partițiile unui număr natural

Fiecare număr natural se poate scrie ca suma altor numere naturale. Pentru toate numerele mai mari decât 1 există mai multe posibilități în acest sens. Scrieți un program care listează toate partițiile unui număr natural ($1 \le n \le 75$) şi apoi numărul de soluții. Exemplu:

n.in	nPart.out
5	1 + 1 + 1 + 1 + 1 1 + 1 + 1 + 2 1 + 1 + 3 1 + 2 + 2 1 + 4 2 + 3 5 Numar de solutii = 7
50	...24 + 26 25 + 25 50 Numar de solutii = 204226
75	...36 + 39 37 + 38 75 Numar de solutii = 8118264

Analiza problemei şi proiectarea soluției

Considerăm că o soluție a problemei este vectorul $x[1]$, $x[2]$, ..., $x[k]$ cu proprietățile:
- elementele sale sunt în ordine crescătoare;
- suma lor este n.

Secvența $x[1]$, $x[2]$, ..., $x[k-1]$ este candidat la o soluție (poate să conducă la construcția unei soluții), dacă elementele sunt în ordine crescătoare şi suma lor este mai mică sau egală cu n. Pentru a calcula $x[k]$, inițializăm $x[k]$ cu $x[k-1]-1$ şi folosim un vector $s[i]$, care conține suma primelor i elemente ale vectorului $x[]$. Atunci când $s[i] = n$, înseamnă că am determinat o soluție a problemei.

Program

```
#include <fstream>
#include <vector>

using namespace std;

void writeSolution(vector<int> x, ofstream &out){
  out << x[0];
  for(int i=1; i<x.size(); i++) out << " + " << x[i];
  out << endl;
}

int main(){
  unsigned long long noSol=0;
  int n, k;
  vector<int> x, s;
  bool isSuccessor, isCandidate;
  ifstream in("n.in");
  ofstream out("nPart.out");
  in >> n; x.push_back(0);
  while(!x.empty()){
    k=x.size();
    do{
      if(x[k-1]<n){
        isSuccessor=true; x[k-1]++;
        if(s.size()==k) s[k-1]=x[k-1];
        else s.push_back(x[k-1]);
        if(k>1) s[k-1]+=s[k-2];
        isCandidate = s[k-1]<=n;
      }else
          isSuccessor=false;
    }while(isSuccessor && !isCandidate);
    if(isSuccessor){
      if(s[k-1]==n) {
        writeSolution(x, out);
        noSol++;
      }
      else
        x.push_back(x[k-1]-1);
      } else{
        x.pop_back();
        if(s.size()==k) s.pop_back();
      }
    }
  out << "Numar de solutii = " << noSol;
  return 0;
}
```

```
// o soluție cu tablouri
...
int isSuccessor,
      isCandidate;
int s[500];
cout << "n = ";
cin  >> n;
k = 1;
x[k] = 0;
s[0]=0;
while(k>0){
 do{
  if(x[k]<n){
   isSuccessor=1;
   x[k]++;
   s[k]=s[k-1]+x[k];
   isCandidate=s[k]<=n;
  }else
   isSuccessor=0;
 }
while
(isSuccesor&&!isCandidate);
 if(isSuccessor)
  if(s[k]==n)Process(x, k)
  else{
    ++k;
    x[k]=x[k-1]-1;
  }
 else k--;
}
...
```

Probleme propuse

1. Scrieți o implementare completă a soluției cu tablouri.
2. Modificați programul astfel încât partițiile să fie listate în ordine colexicografică. Exemplu:

n.in	nPart.out
5	5 2 + 3 1 + 4 1 + 2 + 2 1 + 1 + 3 1 + 1 + 1 + 2 1 + 1 + 1 + 1 + 1 Solution No = 7

3. Modificați programul astfel încât căutarea unui candidat $x[k]$ să se facă cu ajutorul algoritmului din programul *BP* (problema celor n regine) (în loc de variabilele *isSuccessor* şi *isCandidate* folosiți variabila *flag*, iar în locul buclei *do{...}while* bucla *while{}* etc.).
4. Scrieți un program recursiv ce rezolvă problema.
5. Numărul de elemente 1 în toate partițiile este acelaşi cu suma numerelor de elemente diferite în fiecare partiție. Demonstrați această propoziție. Exemplu pentru $n = 5$:

partițiile	numărul elementelor diferite
1 + 1 + 1 + 1 + 1	1
1 + 1 + 1 + 2	2
1 + 1 + 3	2
1 + 2 + 2	2
1 + 4	2
2 + 3	2
5	1
Total:	*12*

Numărul de elemente egale cu 1 în toate partițiile este tot 12. Scrieți un program care verifică propoziția pentru $1 \leq n \leq 40$. Programul trebuie să creeze fişierul *parti.out* de forma:

parti.out
2 2 2 OK ... 5 12 12 OK ... 27 11732 11732 OK 28 14742 14742 OK ... 40 177970 177970 OK

Problema 7. Referate la geografie

Doamna profesoară de geografie a prezentat elevilor o serie de teme pentru referate. Numărul elevilor este n şi numărul temelor este m ($m \leq n$), iar fiecare temă trebuie să fie abordată cel puţin o dată. Scrieţi un program care să listeze toate posibilităţile de a împărţi cele m teme de referate celor n elevi.

Date de intrare. Numerele naturale n şi m se citesc de la tastatură.

Date de ieşire. Scrieţi în fişierul *referat.out* toate modalităţile de împărţire a temelor, ca în exemplu:

Tastatură	referat.out
n=3 m=2	1 1 2 1 2 1 1 2 2 2 1 1 2 1 2 2 2 1 Numar de solutii = 6

Analiza problemei şi proiectarea soluţiei

Definiţie:

O funcţie $f : A \to B$ se numeşte *surjectivă* atunci şi numai atunci când $f(A) = B$ (adică $\forall y \in B \; \exists x \in A : f(x)=y$).

Pentru a rezolva această problemă, trebuie să generăm toate funcţiile surjective cu domeniul $A = \{1, 2, \ldots, n\}$ şi codomeniul $B=\{1, 2, \ldots, m\}$. Aceasta este încă o problemă de tipul *Backtracking liniar cu lungime fixă.* Vom folosi, ca şi mai înainte, vectorul $x[]$ pentru a construi soluţia cu valori posibile şi de fiecare dată verificăm dacă $x[1]$, $x[2]$, ..., $x[k]$ poate conduce la o soluţie a problemei. Mai folosim în plus şi vectorul *mark*[], astfel încât *mark*[i] ne semnalizează de câte ori a fost utilizată valoarea i din B în elementele $x[1]$, ..., $x[k]$. În bucla *while*, unde se caută o valoare încă netestată pentru $x[k]$, luăm în considerare:

- dacă o altă valoare pentru $x[k]$ a fost deja testată (`if(x[k]>=0)...`), atunci numărul de valori $x[k]$ în vectorul *vMark*[] va fi decrementat (`vMark[x[k]]--`);
- în următoarea instrucţiune *if* vom calcula în *noMarked* numărul de elemente diferite care se află în elementele $x[0]$, $x[1]$, ..., $x[k]$;
- atunci când suma numărului *noMarked* şi a numărului de poziţii rămase (n–1–k) este mai mică decât m, atunci variabila *flag* ia valoarea *falsă*; în acest caz, nu putem obţine o soluţie cu elementele fixate $x[0]$, $x[1]$, .., $x[k]$.

Observație

Numărul de funcții surjective cu domeniul $A = \{1, 2, \ldots, n\}$ şi codomeniul $B = \{1, 2, \ldots, m\}$ este:

$$S_j(n,m) = \sum_{i=0}^{m-1} (-1)^i C_m^i (m-i)^n = m^n - C_m^1 \cdot (m-1)^n + \ldots + (-1)^{m-1} \cdot m \quad (1)$$

(acest lucru se poate demonstra, de exemplu, prin inducție sau folosind principiul incluziunii şi excluziunii din combinatorică). Dacă s-ar cere doar numărul de posibilități pentru împărțirea referatelor, atunci putem folosi această formulă.

Program

```
#include <iostream>
#include <fstream>
#include <vector>


using namespace std;


void writeSolution(vector<short> &x, ofstream &fout){
  for(int i=0; i<x.size(); i++)
    fout << x[i]+1 << " ";
  fout << endl;
}


int main(){
  short n, m, k, i, noMarked;
  unsigned long long noSol=0;
  ofstream out("referat.out");
  bool flag;
  vector<short> x, vMark;
  cout << "n = "; cin >> n;
  cout << "m = "; cin >> m;
  for(i=0; i<n; i++) vMark.push_back(0);
  x.push_back(-1);
  while(!x.empty()){
    k = x.size()-1;
    flag = false;
    while(!flag && x[k]<m-1){
      if(x[k]>=0) vMark[x[k]]--;
      vMark[++x[k]]++;
      flag = true; noMarked=0;
```

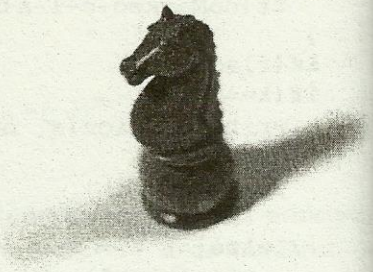
```
        for(i=0; i<=k; i++)
          if(vMark[i]) noMarked++;
        if(noMarked+n-1-k<m)flag=false;
      }
      if(flag){
      if(k==n-1){
        writeSolution(x, out);
        noSol++;
        }else
           x.push_back(-1);
        }else{
          if(x[k]>=0) vMark[x[k]]--;
          x.pop_back();
        }
  }
  out << "Numar de solutii =  " << noSol;
  return 0;
}
```

Probleme propuse

1. Modificați programul astfel încât soluțiile să fie listate colexicografic.
2. Analizați modalitatea de căutare a elementului candidat $x[k]$ în problema anterioară (folosind variabilele *isCandidate* și *isSuccessor*) și folosiți această căutare în program.
3. Implementați o variantă recursivă.
4. Demonstrați formula (1). Găsiți o formulă recursivă pentru determinarea numărului de funcții surjective pe mulțimea $A = \{1, 2, \ldots, n\}$ cu valori în mulțimea $B = \{1, 2, \ldots, m\}$. Scrieți un program care să determine acest număr. Exemplu:

nm.in	noSurj.out
3 2	6
7 4	8400

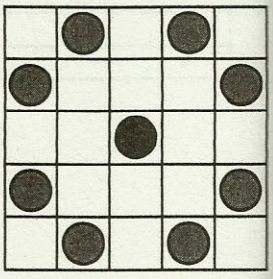
Problema 8. Toate drumurile calului pe tabla de şah



Determinați toate drumurile calului pe o tablă de şah de dimensiune 5 × 5 atunci când este dat câmpul de start. Toate câmpurile trebuie parcurse o singură dată. Fiecare soluție va fi o matrice pătratică de dimensiuni 5 × 5, care simbolizează tabla şi conține numerele de la 1 la n^2 în câmpurile corespunzătoare şi va fi scrisă în fişierul *cal.out*. Poziția de start va fi notată cu 1, iar câmpurile unde calul se deplasează succesiv vor fi numerotate cu 2, 3, ..., 25, ca în exemplu. Dacă pentru câmpul dat nu există nici o soluție, atunci scrieți *„nici o solutie"* în fişierul de ieşire.

Tastatură	cal.out
4 2	Solutia: 1 19 14 3 8 25 2 9 18 13 4 15 20 5 24 7 10 **1** 22 17 12 21 16 11 6 23 ... Solutia: 56 23 12 7 4 21 6 17 22 13 8 11 24 5 20 3 16 **1** 18 9 14 25 10 15 2 19

Analiza problemei şi proiectarea soluției



Acesta este un exemplu clasic pentru *Backtracking în plan*. Dintr-un câmp dat al tablei, putem să ne deplasăm în cel mult opt poziții. Dacă (x, y) sunt coordonatele unui câmp dat, atunci pozițiile succesoare pot fi: $(x - 2, y - 1)$, $(x - 2, y + 1)$, $(x - 1, y - 2)$, $(x - 1, y + 2)$, $(x + 1, y - 2)$, $(x + 1, y + 2)$, $(x + 2, y - 1)$, $(x + 2, y + 1)$. Vom inițializa toate câmpurile tablei de şah cu – 1 şi câmpul de start cu 0. Vom scrie o metodă *Backtracking* recursivă *back*(), ai cărei trei parametri sunt coordonatele câmpului de start şi numărul de câmpuri deja parcurse. Decidem că un câmp (*xnew*, *ynew*) este candidat pentru o soluție atunci când câmpul respectiv se află pe tablă şi dacă valoarea sa este –1 (acest câmp nu a fost încă parcurs). Atunci când un câmp este candidat la o soluție, vom apela metoda *back()*: *back*(*lnew*, *cnew*, *step* + 1).

Program

```
#include <iostream>
#include <cmath>
#include <fstream>

using namespace std;

int n;
int Table[21][21];
int nSol = 0;
ofstream f("cal.out");

writeSolution(){
  f << " Solutia: " << ++nSol << endl;
  for(int i=0; i<n; i++){
    for(int j=0; j<n; j++){
      f.width(3);
      f << Table[i][j] + 1 << "  ";
    }
    f << endl;
  }
  return 0;
}

void back(int l, int c, int step){
  if (n*n-1==step) {writeSolution(); return;}
  int dx, dy, lnew, cnew;
  for(dx = -2; dx<3; dx++)
    for(dy = -2; dy<3; dy++)
     if(abs(dx*dy)==2){
       lnew = l + dx;
       cnew = c + dy;
       if(0<=lnew && lnew<n &&
          0<=cnew && cnew<n &&
          Table[lnew][cnew]==-1){
         Table[lnew][cnew] = step+1;
         back(l+dx, c+dy, step+1);
         Table[lnew][cnew] = -1;
       }
     }
}

int main(){
  int l, c;
  n=5;
  for(l=0; l<n; l++)
```

```
    for(c=0; c<n; c++)
      Table[l][c] = -1;
  cin >> l >> c;
  Table[l-1][c-1] = 0;
  back(l-1, c-1, 0);
  if(0==nSol) f << "nici o solutie";
  return 0;
}
```

Probleme propuse

1. Modificați programul astfel încât să nu mai fie folosite variabile globale.
2. Scrieți o implementare recursivă a algoritmului.
3. Ne imaginăm că unele câmpuri ale tablei sunt închise, adică nu este permisă parcurgerea acestora. Modificați problema şi programul astfel încât să se determine toate drumurile calului, iar câmpurile închise să nu fie parcurse.

Problema 9. Problema fotografiei

O fotografie alb-negru este dată sub forma unei matrice binare (ce conține doar valori 0 şi 1). Ea reprezintă mai multe obiecte. Zonele care aparțin aceluiaşi obiect conțin numai elemente de 1, fundalul conține zerouri. Se cere numărul de obiecte *noOb* şi o nouă reprezentare pentru fotografie, astfel încât obiectele diferite de pe aceasta să fie numerotate cu culori diferite: 2, 3, ..., *noOb* + 1. Doi de 1 care sunt vecini pe linie sau pe coloană în matrice aparțin aceluiaşi obiect.

Date de intrare. În fişierul *foto.in*, se află pe prima linie dimensiunile matricei (a fotografiei), care nu depăşesc 100, urmate de fotografia alb-negru în forma unei matrice binare.

Date de ieşire. Scrieți în fişierul *foto.out* pe prima linie numărul de obiecte, apoi fotografia colorată, astfel încât obiectele să aibă culorile 2, 3, ..., *noOb* + 1, ca în exemplu:

foto.in	foto.out
4 5 1 1 0 1 1 0 0 0 1 0 1 1 0 1 1 1 0 1 1 0	3 2 2 0 3 3 0 0 0 3 0 4 4 0 3 3 4 0 3 3 0

Analiza problemei şi proiectarea soluţiei

Vom scrie metoda recursivă *color()*, care verifică dacă punctul dat ca parametru se află în interiorul fotografiei şi are valoarea 1. În acest caz, colorăm toţi vecinii punctului cu culoarea curentă şi apelăm metoda pentru fiecare vecin.

Program

```
#include <fstream>

using namespace std;
int a[30][30], m, n;

color(int i, int j, int cul){
  if(0<=i && i<m && 0<=j && j<n && 1==a[i][j]){
    a[i][j]=cul;
    color(i, j-1, cul);
    color(i, j+1, cul);
    color(i-1, j, cul);
    color(i+1, j, cul);
  }
}

int main(){
  int i, j, cul;
  ifstream fin("foto.in");
  ofstream fout("foto.out");
  fin >> m >> n;
  for(i=0; i<m; i++)
    for(j=0; j<n; j++)
      fin >> a[i][j];
  cul=1;
  for(i=0; i<m; i++)
    for(j=0; j<n; j++)
      if(1==a[i][j]) color(i, j, ++cul);
  fout << --cul;
  for(i=0; i<m; i++){
    fout<<endl;
    for(j=0; j<n; j++)
      fout << a[i][j] << " ";
  }
  return 0;
}
```

Probleme propuse

1. Modificaţi programul astfel încât să nu mai fie folosite variabile globale.

2. Presupunem că şi două puncte care sunt vecine pe diagonală aparţin aceluiaşi obiect; cu această nouă condiţie în plus, un punct va putea avea maximum opt vecini. Ţineţi cont de aceasta în program.

Problema 10. Rostogolirea mingii

Considerăm o matrice de dimensiuni m şi n, ale cărei elemente sunt înălţimile unităţilor (metri) într-un câmp de joc. Undeva în acest câmp se plasează o minge, care poate să alunece doar pe un câmp vecin cu o înălţime mai mică decât cel pe care se află. Un vecin se află într-unul dintre cele patru puncte cardinale. Găsiţi toate posibilităţile prin care mingea se poate deplasa în acest fel în afara câmpului (ajunge pe o linie sau pe o coloană de margine a matricei). Presupunem că valorile înălţimilor sunt alese în aşa fel, încât întotdeauna există cel puţin o soluţie.

Date de intrare. În fişierul *minge.in*, se găsesc pe prima linie valorile m şi n ($4 \leq m, n \leq 20$), pe următoarele m linii se află înălţimile câmpului, iar pe ultima linie se află coordonatele câmpului de start.

Date de ieşire. Se vor scrie în fişierul *minge.out*. Fiecare linie a fişierului de ieşire conţine câte un drum posibil al mingii către marginea câmpului, ca în exemplu:

minge.in	minge.out
4 4 1 1 0 1 3 2 1 3 4 5 2 3 3 6 1 0 3 2	(3, 2) (2, 2) (1, 2) (3, 2) (2, 2) (2, 3) (1, 3) (3, 2) (3, 1) (3, 2) (3, 3) (2, 3) (1, 3) (3, 2) (3, 3) (4, 3)

Analiza problemei şi proiectarea soluţiei

Soluţia utilizează *Backtracking în plan*. Vom codifica fiecare poziţie cu un număr natural: câmpului (i, j), cu $0 \leq i < m$, $0 \leq j < n$, îi asociem valoarea $k \leftarrow i \cdot n + j$. Pentru o astfel de valoare k, se vor determina linia şi coloana corespunzătoare prin: $i \leftarrow k \ div \ n$, $j \leftarrow k \ mod \ n$. Folosind această codificare, vom putea să lucrăm cu numere naturale în loc de perechi (*linie, coloană*). Implementăm metodele ajutătoare *onTheBorder()* (testează dacă un câmp al matricei se află pe marginea acesteia) şi *onTheTable()* (testează dacă o pereche de numere reprezintă un câmp de pe matrice). Metoda care găseşte toate drumurile este metoda recursivă *back()*. Condiţia de ieşire din metodă este îndeplinită atunci când ultimul element al vectorului $x[]$ se află pe conturul matricei. Atunci când nu ne aflăm pe acest contur, va fi apelată recursiv metoda pentru toţi vecinii câmpului curent care au o înălţime mai mică decât el.

Program

```
#include <fstream>
#include <vector>

using namespace std;

bool onTheBorder(int k, short m, short n){
  short l = k/n;
  short c = k%n;
  bool flag=false;
  if(0==l || m-1==l) flag=true;
  if(0==c || n-1==c) flag=true;
  return flag;
}

bool onTheTable(short line, short col, short m, short n){
  return
    (0<=line && line<m) &&
    (0<=col  &&  col<n);
}

void writeSolution(vector<int> &x, short T[][100],
                   short n, ofstream &out){
  for(int i=0; i<x.size(); i++){
    out << "(" << x[i]/n+1 << ", " << x[i]%n+1 << ") ";
  }
  out << endl;
}

void back(vector<int> x, short T[][100], short m,
          short n, ofstream &out){
  short k = x.size()-1;
  if(onTheBorder(x[k], m, n)){
    writeSolution(x, T, n, out);
    return;
  }
  short l = x[k]/n;
  short c = x[k]%n;
  short lnew, cnew, dx, dy;
  for(dx=-1; dx<2; dx++)
    for(dy=-1; dy<2; dy++)
      if(1==abs(dx+dy)){
        lnew = l+dx;
        cnew = c+dy;
        if(onTheTable(lnew, cnew, m, n)&&T[lnew][cnew]<T[l][c]){
          x.push_back(lnew*n+cnew);
```

```
for(dx=-1; dx<2; dx++)
for(dy=-1; dy<2; dy++)
    if(1==abs(dx+dy))...
```

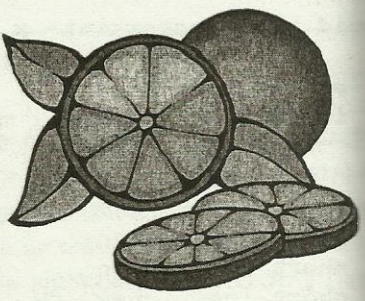
Furnizează pentru *(dx, dy)* valorile (–1, 0), (1, 0), (0, –1), (0, 1), adică direcțiile *N, S, V, E*. Câmpurile vecine sunt așadar *(lnew, cnew)*: *(l – 1, c)*, *(l + 1, c)*, *(l, c – 1)*, *(l, c + 1)*. Când noul element se află pe matrice și înălțimea sa este mai mică decât cea actuală, atunci facem un pas mai departe...

```
            back(x, T, m, n, out);
            x.pop_back();
          }
        }
}

int main(){
  ifstream in("minge.in");
  ofstream out("minge.out");
  short i, j, m, n;
  short l0, c0;
  short T[100][100];
  vector<int> x;
  in >> m >> n;
  for(i=0; i<m; i++)
    for(j=0; j<n; j++)
      in >> T[i][j];
  in >> l0 >> c0;
  l0--; c0--;
  x.push_back(l0*n+c0);
  back(x, T, m, n, out);
  return 0;
}
```

Probleme propuse

1. Dezvoltați mai departe programul astfel încât să fie afișat și numărul soluțiilor și mingea să se poată deplasa și pe diagonale.
2. Scrieți un program iterativ pentru problemă.

Problema 11. Portocalosport



Jocul Portocalosport pare să fie simplu, dar vom vedea că poate deveni problematic. Mai întâi vom numerota 100 de portocale cu o cerneală netoxică de la 1 la 100. Apoi toate vor fi așezate într-un platou uriaș pe mijlocul mesei. Cei doi jucători trebuie să mănânce portocale într-un interval dat de timp (să zicem 15 minute) și să calculeze produsul numerelor scrise pe cele mâncate. Portocala cu numărul 1 nu are voie să fie mâncată împreună cu alte portocale; aceasta înseamnă că unul dintre jucători are voie să o mănânce doar la început și după aceea să nu mai consume altele. Atunci când un jucător raportează la sfârșit numărul 1, atunci a mâncat doar portocala 1, când numărul este mai mare decât 1, atunci nu a mâncat-o sigur. Atunci când timpul de mâncat a expirat, cei doi jucători

trebuie să raporteze produsele numerelor de pe portocalele mâncate (scorurile). Scorurile raportate nu au voie să fie identice (dacă e aşa, atunci jucătorii vor mai fi lăsaţi să mănânce câte o portocală din platou). În mod normal, va câştiga jocul jucătorul care raportează un scor mai mare. Spunem „în mod normal", deoarece fiecare dintre jucători ar fi putut să mintă sau să calculeze greşit produsul. Este simplu de imaginat că se ajunge la controverse, care trebuie mai întâi rezolvate, înainte de a se declara câştigătorul final. Atunci când anumite condiţii sunt satisfăcute, este posibil să câştige şi jucătorul care a declarat un scor mai mic. Presupunem că jucătorul cu scor mai mic spune întotdeauna adevărul atunci când scorul raportat de el se poate descompune ca produs de numere diferite din mulţimea {1, 2, ..., 100}. În acest caz, el poate să câştige atunci când se demonstrează că jucătorul celălalt a mințit. Această minciună poate fi dovedită prin faptul că produsul său se poate descompune în numere de pe portocale care au fost mâncate şi de celălalt jucător. Jucătorul cu scor mai mare poate să câştige atunci când nu se poate demonstra că a mințit, adică există o modalitate de a mânca portocalele care să nu se contrazică cu a celuilalt. Când nici unul dintre rezultate nu se poate descompune cu ajutorul factorilor din {1, 2, ..., 100}, atunci este remiză.

Exemplul 1. Când primul jucător spune 343, iar al doilea 49, atunci primul jucător minte. Singura modalitate de a obţine 343 este să fi mâncat portocalele 7 şi 49. Singura modalitate de a obţine 49 este să fi mâncat portocala 49. De aici, deducem că primul jucător minte (al doilea spune adevărul prin definiţie!).
Exemplul 2. Când primul jucător spune 194 şi al doilea 178, atunci singura modalitate pentru primul jucător este să fi mâncat portocalele 2 şi 97. Al doilea jucător are doar posibilitatea 2 şi 89. Pentru că al doilea spune adevărul prin definiţie, rezultă că primul minte, deci al doilea câştigă (nu se putea să mănânce amândoi portocala 2!).
Exemplul 3. Scoruri declarate: 138 şi 258. Pentru a obţine 138, avem posibilităţile (6, 23), (3, 46), (2, 69) şi (2, 3, 23). Putem presupune că primul a mâncat portocalele 6 şi 23, iar al doilea portocalele 3 şi 86. Atunci al doilea a spus adevărul şi câştigă jocul.
Exemplul 4. Scoruri declarate: 1236 şi 100. Nu există nici o posibilitate de a obţine 1236, deci primul jucător minte. Pentru a obţine 100, avem posibilităţile (100), (5, 20), (4, 25), (2, 50) şi (2, 5, 10), deci al doilea spune adevărul şi câştigă jocul.

Din păcate, toţi care au luat parte la joc au mâncat atât de multe portocale, încât nu mai sunt în stare să facă astfel de calcule complicate. Scrieţi un program care să rezolve problema automat.
Date de intrare. În fişierul *porto.in* se găsesc una sau mai multe perechi de scoruri declarate, câte una pe linie. Presupunem că toate scorurile se încadrează în tipul *unsigned long long*.

Date de ieşire. Pentru fiecare pereche din fişierul de intrare scrieți rezultatul pe o linie a fişierului *porto.out*, care poate fi una dintre propozițiile:

```
Castiga primul jucator!
Castiga al doilea jucator!
Primul s-a incurcat! Castiga al doilea!
Al doilea s-a incurcat! Castiga primul!
Amandoi s-au incurcat! Remiza!
```

Primele două propoziții vor fi scrise atunci când ambele rezultate se pot descompune în factori diferiți între 1 şi 100, iar jucătorul cu scor mai mare câştigă atunci când nu a mințit. A treia şi a patra propoziție se vor scrie atunci când scorul declarat de exact unul dintre jucători nu se poate descompune ca produs de numere diferite dintre 1 şi 100. Ultima propoziție se va afişa atunci când ambele scoruri declarate nu pot fi obținute conform regulilor. Exemplu:

porto.in	porto.out
110 119	Castiga al doilea jucator!
294 202	Al doilea s-a incurcat! Castiga primul!
343 49	Castiga al doilea jucator!
610 3599	Castiga primul jucator!
138 258	Castiga al doilea jucator!
941 2234	Amandoi s-au incurcat! Remiza!
1236 100	Primul s-a incurcat! Castiga al doilea!
6231 1500	Castiga primul jucator!
151 127	Amandoi s-au incurcat! Remiza!
1 101	Al doilea s-a incurcat! Castiga primul!

(inspirată de ACM South-Central USA Collegiate Programming Contest, 1998, Gizilc...)

Analiza problemei şi proiectarea soluției

Problema se reduce la determinarea tuturor posibilităților de a descompune un număr natural n dat, ca produs de numere din mulțimea {1, 2, ..., 100}. Atunci când toate aceste descompuneri pentru ambii jucători sunt identificate, le comparăm două câte două pentru a decide dacă există o pereche de descompuneri disjuncte.

Pentru un număr natural n dat, toate descompunerile ca produs de numere diferite din mulțimea {1, ..., 100} vor fi salvate într-un vector de mulțimi (de tipul *TSetsVector*). Metoda *divVect(ULLINT n, vector<short>& a)* furnizează în vectorul a toți divizorii lui n din mulțimea {2, ..., 100}. Acest vector a va fi folosit de metoda *Backtracking* recursivă *back()*. Această metodă construieşte vectorul x ca fiind vectorul caracteristic al unei submulțimi din a ($x[k] = 1$, dacă elementul k din a este introdus în soluție; $x[k] = 0$, dacă elementul k din a nu este introdus în soluție). Metoda funcționează astfel:

- produsul elementelor din soluția parțială curentă se salvează în p;

- dacă $p = n$, atunci am determinat o descompunere pentru n, care va fi introdusă în vectorul cu mulțimi cu ajutorul metodei *addSet()*;
- dacă $p \neq n$, atunci setăm valoarea $x[k]$ mai întâi *false* și apoi *true*;
- $x[k]$ ia valoare *false*, aceasta înseamnă că divizorul k este introdus în soluție, apoi apelăm metoda *back()* pentru următorul nivel $(k + 1)$;
- $x[k]$ ia valoare *true*, al k-lea divizor ($a[k-1]$ în program) poate să fie adăugat soluției parțiale numai dacă $(p \cdot a[k-1])$ este un divizor al lui n (poate conduce la obținerea unei descompuneri valide). O altă condiție este ca $\frac{n}{p \cdot a[k-1]} = 1$ ($a[k-1]$ va fi ultimul divizor în descompunere) sau $\frac{n}{p \cdot a[k-1]} > a[k-1]$ (elementele descompunerii sunt considerate în ordine crescătoare):

```
if((n/p)%a[k-1]==0){                          //n divizibil cu p·a[k-1]
   x[k]=1;
   if((n/p)/a[k-1]==1 || (n/p)/a[k-1]>a[k-1])   {...}...
```

Atunci când o descompunere validă a fost găsită, aceasta va fi adăugată vectorului b cu soluții, cu ajutorul metodei:

```
void addSet(int k, TSetsVector& b) {...}
```

Vectorul b complet cu soluții corespunzătoare parametrului n se va construi cu ajutorul metodei:

```
void buildSetsVector(ULLINT n, TSetsVector& b){
  divVect(n, a);
  b.clear(); x.reset();
  back(n, 1, b);
}
```

Pentru punctajele citite, *n1* și *n2*, se vor construi vectorii cu descompuneri b și c în programul principal cu ajutorul:

```
buildSetsVector(n1, b);
buildSetsVector(n2, c);
```

Metoda *makeDecision()* stabilește care este rezultatul jocului pe baza vectorilor de descompuneri b și c. Ea folosește funcția *isADisjointCombination(TSetsVector b, TSetsVector c)*, care decide dacă există o descompunere în b și o descompunere în c care sunt disjuncte (mulțimile corespunzătoare au intersecția vidă).

Remarcăm utilizarea tipului *bitset<N>* pentru stocarea vectorului caracteristic *x* în metoda recursivă *Backtracking*. Un astfel de element se diferenţiază de tipul *vector<bool>* prin faptul că are lungime fixă.

Program

```
#include <fstream>
#include <set>
#include <vector>
#include <bitset>

using namespace std;

typedef unsigned long long ULLINT;
typedef set<short> TSet;
typedef vector<TSet> TSetsVector;

vector<short> a;
bitset<30> x;

void addSet(int k, TSetsVector& b) {
  int i, t=1;
  TSet auxSet;
  for(i=1; i<k; i++)
    if(x[i])
      auxSet.insert(a[i-1]);
  b.push_back(auxSet);
}

void back(ULLINT n, int k, TSetsVector& b){
  int i, j;
  long int p = 1;
  int aSiz = (int)a.size();
  for(j=1; j<k; j++) if(x[j]) p*=a[j-1];
  if(p == n) {
    addSet(k, b);
  }
  else
    if(k <= aSiz)
      for(i=0; i<2; i++){
        x[k] = 0==i ? false:true;
        if(!i) back(n, k+1, b);
        else
          if(0==(n/p)%a[k-1]){
            x[k]=true;
            if(1==(n/p)/a[k-1] || (n/p)/a[k-1]>a[k-1])
              back(n, k+1, b);
```

```
            }
        }
}

void divVect(ULLINT n, vector<short>& a){
  a.clear();
  for(int i=2; i<=100; i++)
    if(0==n%i)
      a.push_back(i);
}

void buildSetsVector(ULLINT n, TSetsVector& b){
  divVect(n, a);
  b.clear(); x.reset();
  back(n, 1, b);
}

bool isACommonElement(TSet A, TSet B){
  bool flag = false;
  set<short>::const_iterator itA;
  for (itA = A.begin(); !flag && itA != A.end(); itA++)
    if (B.find(*itA) != B.end()){
      flag = true;
    }
  return flag;
}

bool isADisjointCombination(TSetsVector b, TSetsVector c){
  bool flag = false;
  short bSiz = (short) b.size(), cSiz = (short)c.size();
  for(short i=0; !flag && i<bSiz; i++)
    for(short j=0; !flag && j<cSiz; j++)
      if(!isACommonElement(b[i], c[j])) flag=true;
  return flag;
}

void makeDecision(ULLINT n1, ULLINT n2,
                  TSetsVector b, TSetsVector c,
                  ofstream &fout){

  if(!b.size() && !c.size())
    fout << "Amandoi s-au incurcat! Remiza!" << endl;
  else if(!b.size())
    fout << "Primul s-a incurcat! Castiga al doilea!" << endl;
  else if(!c.size())
    fout << "Al doilea s-a incurcat! Castiga primul!" << endl;
  else
    if(n1<n2)
```

```
        if(isADisjointCombination(b, c))
          fout << "Castiga al doilea jucator!" << endl;
        else
          fout << "Castiga primul jucator!" << endl;
      else
        if (isADisjointCombination(b, c))
          fout << "Castiga primul jucator!" << endl;
        else
          fout << "Castiga al doilea jucator!" << endl;
}

int main(){
  ULLINT n1, n2;
  TSetsVector b, c;
  ifstream fin("porto.in");
  ofstream fout("porto.out");
  while(fin && !fin.eof() && fin>>n1>>n2){
    buildSetsVector(n1, b);
    buildSetsVector(n2, c);
    makeDecision(n1, n2, b, c, fout);
  };
  fin.close(); fout.close();
  return 0;
}
```

Probleme propuse

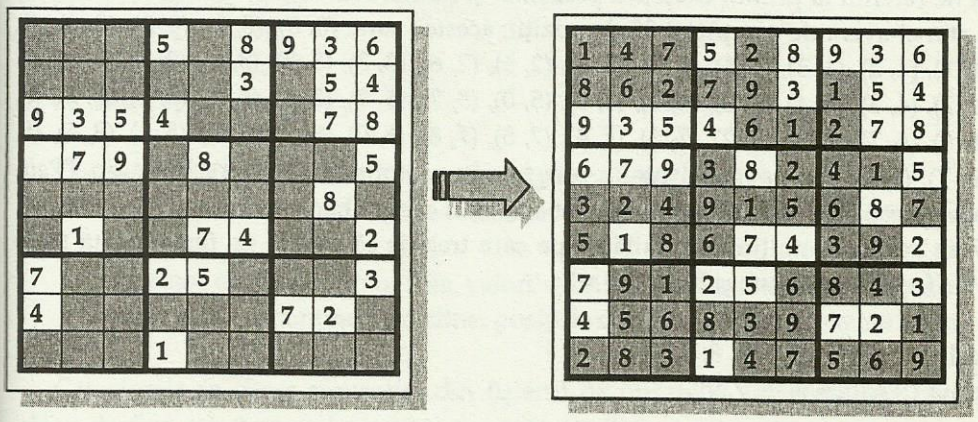
1. Modificați programul astfel încât să nu mai fie utilizate variabile globale.
2. Atunci când un jucător câştigă, furnizați şi o posibilitate ca el să fi mâncat porto-calele (când este posibil, câte una pentru fiecare jucător).
3. Scrieți un program C care să rezolve problema (de exemplu, stocați toate descompunerile într-o matrice bidimensională).
4. Citiți în *Help* despre clasa *bitset<N>* din *STL*: descriere, metode, operatori şi exemple.
5. Containerele asociative sunt cele în care informațiile sunt stocate ordonat. Citiți în *Help* despre containerele asociative *std::set* (conceptul de *mulțime* din matematică, în care fiecare element este luat o singură dată) şi *std::multiset* (unele elemente pot să se repete).
6. În *header*-ul *<algorithm>* din *STL* se găsesc şi metode corespunzătoare operațiilor cu mulțimi: *includes* (decide dacă o mulțime *A* este inclusă în altă mulțime *B*), *set_union* (furnizează reuniunea a două mulțimi), *set_intersection* (intersecția) şi *set_symmetric_difference* (diferența simetrică a două mulțimi *A* şi *B*, definită ca $A \Delta B = (A-B) \cup (B-A)$). Modificați programul de mai sus astfel încât în locul meto-dei *isACommonElement(TSet A, TSet B)* să folosiți metoda *set_intersection*. Scrieți

programe demonstrative implementând toate cele patru metode pentru operațiile pe mulțimi. Scrieți funcții echivalente pentru ele (presupunând că acestea nu există).

7. Scrieți o metodă *Backtracking* iterativă în locul celei recursive.
8. Implementați o metodă care furnizează pentru un număr natural (*unsigned long long*) toate descompunerile ca produs de factori diferiți din mulțimea {1, 2, ..., 100}.

Problema 12. Sudoku

Cine nu a auzit de popularul joc de gândire Sudoku? Predecesoare timpurii ale jocului pot fi considerate, de exemplu, *pătratele latine* ale celebrului matematician Leonhard Euler (1707-1783), care a prezentat o serie încă în secolul al XVIII-lea. El le numea *carré latin* şi nu erau împărțite şi în subpătrate, ca în cunoscutele careuri Sudoku de astăzi. Jocul actual se compune dintr-un careu pătratic cu nouă linii şi nouă coloane, care se poate împărți şi în 3 × 3 blocuri, fiecare cu dimensiunea de 3 × 3 câmpuri. În unele câmpuri se găsesc deja cifre de la 1 la 9; în mod normal, sunt deja completate între 22 şi 36 de câmpuri din cele 81. Scopul este acela de a fi completate şi câmpurile libere cu cifre de la 1 la 9, astfel încât pe toate liniile, coloanele şi blocurile fiecare cifră să apară doar o singură dată. Atunci când un număr este posibil într-un anumit câmp, acesta este numit „candidat". Cele trei zone (linie, coloană, bloc) sunt numite „unități". Chiar dacă de obicei Sudoku conține cifre, nu sunt necesare nici un fel de calcule, în locul acestora putând fi folosite orice simboluri. Exemplu:



Exemplu de puzzle Sudoku

Problema cere rezolvarea unui Sudoku care se află în fișierul *sudoku.in*, în care câmpurile libere sunt marcate cu „*". Rezultatul se va scrie în *sudoku.out*, unde se vor lista toate posibilitățile. Dacă nu există soluție, se va scrie *nu există solutie*. Exemplu:

sudoku.in	sudoku.out
***5*8936	1 4 7 5 2 8 9 3 6
*****3*54	8 6 2 7 9 3 1 5 4
9354***78	9 3 5 4 6 1 2 7 8
679*8***5	6 7 9 3 8 2 4 1 5
********8*	3 2 4 9 1 5 6 8 7
*1**74**2	5 1 8 6 7 4 3 9 2
7**25***3	7 9 1 2 5 6 8 4 3
4******72*	4 5 6 8 3 9 7 2 1
1**	2 8 3 1 4 7 5 6 9
*1***94**	8 1 5 7 3 9 4 2 6
*74***39*	6 7 4 2 8 5 3 9 1
3******5**	3 9 2 4 1 6 5 8 7
2***5*673	2 8 1 9 5 4 6 7 3
7***63*4*	7 5 9 8 6 3 1 4 2
628**	4 3 6 1 7 2 8 5 9
*********	9 2 3 5 4 1 7 6 8
7*235	1 4 7 6 2 8 9 3 5
568*9**1*	5 6 8 3 9 7 2 1 4

Analiza problemei şi proiectarea soluției

Dacă ne referim la primul exemplu prezentat şi numerotăm liniile şi coloanele de la 0 la 8, atunci avem de completat 39 de poziții; acestea sunt: (0, 0), (0, 1), (0, 2), (0, 4), (1, 0), (1, 1), (1, 2), (1, 3), (1, 4), (1, 6), (2, 4), (2, 5), (2, 6), (3, 3), (3, 5), (3, 6), (3, 7), (4, 0), (4, 1), (4, 2), (4, 3), (4, 4), (4, 5), (4, 6), (4, 8), (5, 0), (5, 2), (5, 3), (5, 6), (5, 7), (6, 1), (6, 2), (6, 5), (6, 6), (6, 7), (7, 1), (7, 2), (7, 3), (7, 4), (7, 5), (7, 8), (8, 0), (8, 1), (8, 2), (8, 4), (8, 5), (8, 6), (8, 7), (8, 8). Ele vor fi salvate în program în vectorul *poz* cu perechi de tip *TPair*, fiecare pereche reprezentând linia şi coloana unui câmp liber.

Vom construi mulțimi cu unitățile pe care trebuie să se afle de fiecare dată toate cele 9 cifre: linii, coloane şi blocuri 3 × 3:

Linia 1: $sl[0] = \{3, 5, 6, 8, 9\}$
Coloana 1: $sc[0] = \{4, 6, 7, 9\}$

Linia 2: $sl[1] = \{3, 4, 5\}$
Coloana 2: $sc[1] = \{1, 3, 7\}$

Linia 3: $sl[2] = \{3, 4, 5, 7, 8, 9\}$
Coloana 3: $sc[3] = \{5, 9\}$

Linia 4: $sl[3] = \{5, 6, 7, 8, 9\}$
Coloana 4: $sc[3] = \{1, 2, 4, 5\}$

Linia 5: $sl[4] = \{8\}$
Coloana 5: $sc[4] = \{5, 7, 8\}$

Linia 6: $sl[5] = \{1, 2, 4, 7\}$
Coloana 6: $sc[5] = \{3, 4, 8\}$

Linia 7: $sl[6] = \{2, 3, 5, 7\}$
Coloana 7: $sc[6] = \{7, 9\}$

Linia 8: $sl[7] = \{2, 4, 7\}$
Coloana 8: $sc[7] = \{2, 3, 5, 7, 8\}$

			5		8	9	3	6
					3		5	4
9	3	5	4				7	8
6	7	9		8				5
							8	
	1			7	4			2
7			2	5				3
4						7	2	
			1					

Mulțimile corespunzătoare celor patru blocuri sunt:
(0, 0): $sp[0][0] = \{3, 5, 9\}$
(0, 1): $sp[0][1] = \{3, 4, 5, 8\}$
(0, 2): $sp[0][2] = \{3, 4, 5, 6, 7, 8, 9\}$
(1, 0): $sp[1][0] = \{1, 6, 7, 9\}$
(1, 1): $sp[1][1] = \{4, 7, 8\}$
(1, 2): $sp[1][2] = \{2, 5, 8\}$
(2, 0): $sp[2][0] = \{4, 7\}$
(2, 1): $sp[2][1] = \{1, 2, 5\}$
(2, 2): $sp[2][2] = \{2, 3, 7\}$

Declarăm variabilele globale *sl*, *sc*, *sp*, *t* și *poz* cu semnificațiile:

- *sl*[], *sc*[]: tablouri unidimensionale ce conțin mulțimile de cifre aflate pe linii, respectiv coloane;
- *t*: vector ce conține, în ordine, valorile din câmpurile completate inițial;
- *poz*: vector ce conține, în ordine, pozițiile câmpurilor necompletate inițial.

Caracterele se citesc secvențial din fișierul de intrare și se procesează corespunzător: dacă *ch* este *, atunci perechea formată din linia și coloana curente se adaugă la vectorul *poz*; dacă este cifră, atunci valoarea sa se adaugă mulțimilor liniei, coloanei și blocului curent. După citirea unui astfel de caracter valid, se actualizează linia și coloana curente (care sunt inițializate cu 0): dacă se ajunge la coloana 9 (*DIM*), atunci ne deplasăm pe linia următoare și inițializăm coloana cu 0: `l++; c=0;`.

```
if(isdigit(ch) || V==ch){
      if(V==ch) poz.push_back(TPair(l, c));
      else{
        sl[l].insert(ch-'0');
        sc[c].insert(ch-'0');
        sp[l/NR][c/NR].insert(ch-'0');
        t.push_back(ch-'0');
      }
      if(DIM==++c) {l++; c=0;}
}
```

Vom codifica soluția ca un vector v cu aceeași lungime ca și *poz*, cu semnificația: $v[i]$ este cifra care se scrie la poziția $poz[i]$.

Vom scrie o metodă de tip *Backtracking* recursiv *back*(): pentru nivelul k, se verifică dacă avem vectorul v complet, caz în care se face afișarea. Altfel i se vor atribui lui $v[k]$ toate valorile posibile dintre 1 și 9 și pentru fiecare dintre ele se verifică dacă poate conduce la o soluție cu valorile $v[0]$, $v[1]$, ..., $v[k-1]$ folosind metoda *isCandidat*() deja fixate. Dacă $v[k]$ este candidat la soluție, atunci valoarea $v[k]$ se adaugă mulțimilor corespunzătoare: linia $v[k]$.*first*, coloana $v[k]$.*second* și blocul [$poz[k]$.*first*/*NR*] [$poz[k]$.*second*/*NR*], după care se apelează metoda recursivă pentru următorul nivel și se elimină $v[k]$ din mulțimile unde s-a adăugat (prin asignarea unei noi valori pentru $v[k]$ nu mai avem nevoie de valoarea precedentă!).

Scrierea soluției se va face cu ajutorul metodei *writeSolution*(): câmpurile careului se vor parcurge în ordine și cele din *poz* se vor asigna cu valorile din vectorul v, iar cele inițial completate se vor asigna cu valorile din vectorul t (indicii k și j parcurg secvențial cei doi vectori).

Program

```
#include <set>
#include <vector>
#include <fstream>
#include <algorithm>

using namespace std;

const char    V = '*';
const short DIM = 9;
const short  NR = 3;

set<int> sl[DIM], sc[DIM];
set<int> sp[DIM/NR][DIM/NR];

vector<short> t;
typedef pair<short, short> TPair;
```

```
vector<TPair> poz;

bool isCandidat(vector<int> v, int k){
  int ll = poz[k].first;
  int cc = poz[k].second;
  if(sl[ll].find(v[k])!= sl[ll].end()) return false;
  if(sc[cc].find(v[k])!=sc[cc].end())  return false;
  if(sp[ll/NR][cc/NR].find(v[k])!=sp[ll/NR][cc/NR].end())
                                       return false;
  return true;
}

void writeSolution(vector<int> v, ofstream& out){
  int k=0, j=0;;
  int ll=0, cc=0;
  while(ll<DIM){
    if(poz[k].first==ll && poz[k].second==cc){
      out<<v[k++]<<" ";
    }
    else out<<t[j++]<<" ";
    if(++cc==DIM){ll++; cc=0; out<<endl;}
  }
  out<<endl;
}

void back(int k, vector<int>& v, ofstream& out){
  if(k==poz.size()) { writeSolution(v, out); return;}
  for(int i=1; i<=DIM; ++i){
    v[k]=i;
    if(isCandidat(v, k)){
      sl[poz[k].first].insert(v[k]);
      sc[poz[k].second].insert(v[k]);
      sp[poz[k].first/NR][poz[k].second/NR].insert(v[k]);
      back(k+1, v, out);
      sl[poz[k].first].erase(v[k]);
      sc[poz[k].second].erase(v[k]);
      sp[poz[k].first/NR][poz[k].second/NR].erase(v[k]);
    }
  }
}

int main(){
  ifstream in("sudoku.in");
  short l=0, c=0;
  char ch;
  while(in && !in.eof() && in>>ch && l<DIM){
    if(isdigit(ch) || V==ch){
      if(V==ch) poz.push_back(TPair(l, c));
```

```
        else{
          sl[l].insert(ch-'0');
          sc[c].insert(ch-'0');
          sp[l/NR][c/NR].insert(ch-'0');
          t.push_back(ch-'0');
        }
        if(DIM==++c) {l++; c=0;}
      }
    }

    vector<int> v(poz.size(), 0);
    ofstream out("sudoku.out");
    back(0, v, out);

    return 0;
}
```

Probleme propuse

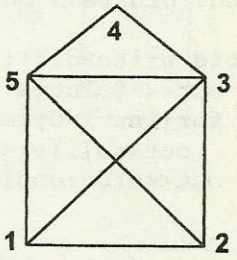
1. Valoarea de pe un câmp inițial completat (l, c) este intersecția mulțimilor liniei l $sl[l]$ și a coloanei c $sc[c]$. Folosiți această observație și eliminați din program utilizarea tabloului t (în metoda *writeSolution*() se va scrie în loc de $t[j++]$ valoarea $sl[ll] \cap sc[cc]$ – aceasta trebuie să fie o mulțime formată dintr-un singur element).
2. Modificați programul astfel încât să nu mai folosească variabile globale.
3. Pentru cazul în care nu există nici o soluție pentru fișierul de intrare dat, fișierul de ieșire *sudoku.out* nu va conține nimic. Modificați programul, astfel încât să scrie în această situație mesajul *„nu exista soluție"*. Exemplu:

sudoku1.in	sudoku1.out
5**92**71 6**15*3** *24*7**8* *51*****7 *4**95*** ***6***** 915*6***4 *6******3 23**4*8**	nu exista solutie

4. Implementați o metodă de tip *Backtracking* iterativ.
5. Scrieți o metodă care verifică dacă un careu complet dat respectă condițiile Sudoku (este corect).
6. Paginile *Web http://www.sachsentext.de/en/index.htm, http://www.maa.org/editorial/mathgames/mathgames_09_05_05.html* conțin mai multe variante ale jocului Sudoku (de exemplu, *Shidoku, Rokudoku, Maxi Sudoku, Irregular Sudoku* etc.). Scrieți

programe şi pentru alte variante; găsiţi eventual un algoritm generalizat pentru rezolvarea mai multor tipuri de Sudoku.

Problema 13. Căsuţa lui Moş Crăciun



Copiii (uneori şi adulţii) desenează de generaţii căsuţa lui Moş Crăciun. Adică acea căsuţă obţinută prin desenarea fără a se ridica de pe hârtie creionul şi fără a trasa de două ori o linie. Acest lucru este posibil doar atunci când se porneşte dintr-unul dintre vârfurile de jos ale căsuţei.

Trebuie să scrieţi un program care să listeze toate posibilităţile de a desena casa, atunci când se porneşte din colţul din stânga-jos (1). Colţurile se vor numerota ca în figură. O posibilitate ar fi, de exemplu, „153125432", ceea ce înseamnă că se porneşte din colţul 1, se desenează o linie până la 5, apoi până la 3 etc.

Soluţiile numerotate şi în ordine lexicografică se vor scrie în fişierul *moscraciun.out*. Exemplu:

moscraciun.out
Solutia 1: 1 2 3 1 5 3 4 5 2
.....
Solutia 26: 1 3 5 2 3 4 5 1 2
.....
Solutia 44: 1 5 4 3 5 2 3 1 2

Analiza problemei şi proiectarea soluţiei

Casa lui Moş Crăciun este, de fapt, reprezentată ca un graf neorientat, în care trebuie să găsim ciclurile euleriene. Pentru aceasta, vom folosi din nou un algoritm de tip *Backtracking* recursiv. Construim succesiv soluţia în tabloul $b[]$; când a fost stabilită o valoare candidat pentru $b[k]$, vom şterge muchia „desenată" ($b[k–1]$, $b[k]$), vom apela metoda recursiv şi apoi vom introduce muchia din nou în graf.

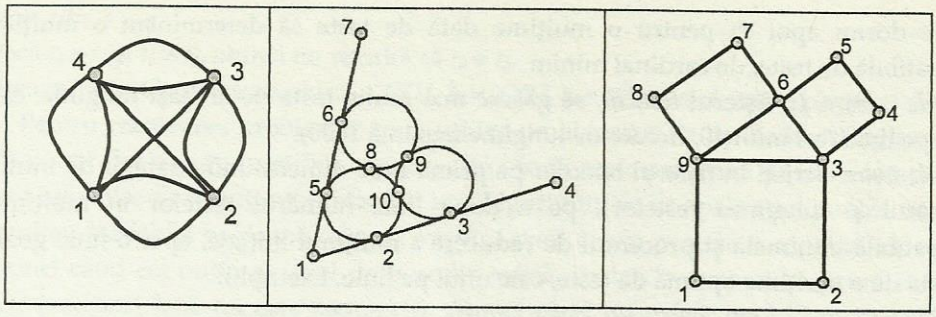
Program

```
#include <fstream>

short a[5][5]={0,1,1,0,1,
```

```
                1,0,1,0,1,
                1,1,0,1,1,
                0,0,1,0,1,
                1,1,1,1,0};
int b[9];
int sol=0;
std::ofstream out("moscraciun.out");

void writeSol(){
  out<<"Solutia " << ++sol << ": ";
  for(int i=0;i<9;i++)
    out<<b[i]+1<<" ";
  out<<std::endl;
}

void back(int k){
  int i;
  if(9==k) writeSol();
  else
    for(i=0;i<5;i++)
      if(a[i][b[k-1]]==1 && i!=b[k-1]){
        b[k]=i;
        a[i][b[k-1]]=0;
        a[b[k-1]][i]=0;
        back(k+1);
        a[i][b[k-1]]=1;
        a[b[k-1]][i]=1;
    }
}

int main(){
  b[0]=0;
  back(1);
  return 0;
}
```

Probleme propuse

1. Modificaţi programul astfel încât să nu mai folosească variabile globale.
2. Scrieţi un algoritm de tip *Backtracking* iterativ.
3. Implementaţi programe şi pentru următoarele figuri, pentru a le desena ca mai sus. La a doua figură se începe cu vârful 4, la prima şi la a treia cu vârful 1.



Problema 14. Compactarea şabloanelor de test

Se dă o mulţime de teste T; se cere determinarea unei mulţimi de teste T' de cardinal minim, astfel încât pentru fiecare test din T să existe un test compatibil în T'. Un test este un şir ce conţine caractere din mulţimea S = {'0', '1', 'U', 'Z', 'X'}. Vom numi S mulţimea de compactare. Exemple de teste: *X01XUZZX01*, *100011ZUX1ZXXX*.

Definiţia 1. Caractere compatibile
Spunem că două caractere sunt compatibile atunci când ambele coincid sau când cel puţin unul dintre ele este X (*don't care*). Vom nota această relaţie cu „≡" şi incompatibilitatea cu „≠". Atunci când cele două caractere coincid, le vom uni în unul singur (*merge*), când unul dintre ele este X, atunci rezultatul operaţiei *merge* va fi celălalt caracter (care nu este X). În tabelul următor sunt prezentate relaţiile compatibil şi *merge* (* înseamnă că nu există compatibilitate).

≡	0	1	U	Z	X
0	0	*	*	*	0
1	*	1	*	*	1
U	*	*	U	*	U
Z	*	*	*	Z	Z
X	0	1	U	Z	X

Definiţia 2. Teste compatibile
Două teste date sunt compatibile atunci când au aceeaşi lungime şi fiecare caracter dintr-un test este compatibil cu caracterul corespunzător din testul celălalt. Din două teste compatibile se creează un test fuzionat (*merged*), în care, pe fiecare poziţie, se află caracterul fuzionat din cele două caractere corespunzătoare în cele două teste. Exemplu: testele t_1 = *10ZX0XU* şi t_2 = *X0Z10UU* sunt compatibile, deoarece $t_1(i) \equiv t_2(i)$, pentru toţi $i \in \{1, 2, \ldots, 7\}$. *Merge*($t_1$, t_2) = *10Z10UU*.

Ne dorim apoi ca pentru o mulțime dată de teste să determinăm o mulțime compatibilă de teste, de cardinal minim.

Date de intrare. În fișierul *tests.in*, se găsesc mai multe teste de aceeași lungime, câte unul pe linie (cel mult 40, fiecare de lungime maximă 1000).

Date de ieșire. Scrieți în fișierul *tests.out* pe prima linie dimensiunile datelor de intrare (numărul și lungimea testelor), pe a doua linie numărul testelor în mulțimea compatibilă minimală și procentul de reducere a mulțimii inițiale, apoi o linie goală, urmată de o mulțime optimă de teste, câte unul pe linie. Exemplu:

tests.in	tests.out
UU0XXZZU	15 8
XUU0XX11	5 33.3333%
XXUXZXXX	
UUU0ZZ11	UU0X1ZZU
0XX111XX	011111ZZ
XUU0XX11	U0UXZU0U
U0XXXUXU	0UUZZ11U
XXXX1XXX	UUU0ZZ11
UXXXXZZU	
011111ZZ	
0XX1XXXX	
XXUXZXXX	
U0UXZU0U	
XUUXZX1X	
0UXZX11U	

(propusă de Rolf Drechsler, Görschwin Fey, Daniel Große)

Analiza problemei și proiectarea soluției

Din definiția problemei rezultă următoarele trei propoziții, care ne vor ajuta în găsirea soluției.

Propoziția 1

Dacă $t_0 \neq t_1$ (nu sunt compatibile), $t_0 \equiv t_2$ (compatibile) și $Merge(t_0, t_2) = t_3$, atunci $t_3 \neq t_1$.

Demonstrație. Deoarece t_0 și t_1 nu sunt compatibile, rezultă că există cel puțin o poziție k, astfel încât $t_0[k] \neq t_1[k]$. După operația de fuzionare, obținem $Merge(t_0, t_2) = t_3$ și rezultă că $t_3[k] = t_0[k]$ și $t_3[k] \neq t_1[k]$, deoarece nici unul dintre ele nu este X. Urmează că t_1 și t_3 nu sunt compatibile. ❑

Propoziția 2

Dacă $t_0 \equiv t_1$ și $Merge(t_0, t_1) = t_2$, $t_2 \equiv t_3$ și $Merge(t_2, t_3) = t_4$, atunci $t_0 \equiv t_4$ și $t_1 \equiv t_4$.

Demonstrație. Dacă un caracter în t_0 este X, atunci el este compatibil cu caracterul de pe aceeași poziție în t_4. Dacă nu este X, atunci prin operația *Merge* a fost creat același caracter și în t_4. ❑

Propoziția 3

Dacă $t_0 \equiv t_1$ și $t_1 \equiv t_2$, atunci nu rezultă că $t_0 \equiv t_2$.

Demonstrație. Contraexemplu: $t_0 = 1XU$, $t_1 = XZU$, $t_2 = 0ZX$ (vezi prima poziție!). ❑

Pentru rezolvarea problemei, vom utiliza un algoritm de tip *Backtracking* recursiv. Un vector dat v se va modifica repetat, astfel încât toate perechile de teste compatibile sunt înlocuite cu rezultatul fuzionării lor. Algoritmul se va apela apoi pentru vectorul rezultat. Variabila booleană *compact* verifică dacă există perechi compatibile de teste. Atunci când cel puțin o astfel de pereche există, variabila *compact* va primi valoarea *false* (vectorul încă nu este compact!). Atunci când nu există perechi compatibile în vector, compact va rămâne *true*, iar v este una dintre soluțiile posibile. În acest caz, trebuie să mai verificăm dacă vectorul curent v este de dimensiune mai mică decât soluția cea mai bună găsită până acum, salvată în vectorul *vOut*. În acest caz, se va înlocui *vOut* cu v.

Algoritmul în pseudocod:

```
ALGORITHM_OPTIM_COMPACT_REC( vector v, vector& vOut )
        bool compact
        compact ← true
        For (i = 1, size(v) – 1; step 1) Execute
                For( j = i + 1, size(v); step 1) Execute
                 If(ti compatible tj) Then
                        compact ← false
                        t ← merge(ti, tj)
                        v.delete(ti)
                        v.delete(tj)
                        v.add(t)
                        ALGORITHM_OPTIM_COMPACT_REC(v, vOut)
                 End_If
                End_For
        End_For
        If (compact) Execute
                If( size(vOut) > size(v) ) Then
                     vOut ← copy_of( V )
                 End_If
         return;
        End_If
END_ALGORITHM _OPTIM_COMPACT_REC
```

Corectitudinea algoritmului. Algoritmul este corect, pentru că vectorul v este în așa fel manipulat încât proprietățile de compatibilitate se păstrează (înlocuirea a două teste compatibile cu testul lor fuzionat).

Complexitatea algoritmului. Arborele apelurilor recursive are înălțimea maximă n (numărul testelor în fișierul de intrare). În acest caz, numărul total de comparații este:

$$\frac{n(n-1)}{2}\cdot\frac{(n-1)(n-2)}{2}\cdot\frac{(n-2)(n-3)}{2}\cdot\ldots\cdot\frac{2\cdot 1}{2}=\prod_{k=2}^{n}\frac{k(k-1)}{2}=\frac{n!\cdot(n-1)!}{2^{n-1}}.$$

De unde rezultă complexitatea algoritmului: $O(\frac{n!\cdot(n-1)!}{2^n})$. ❑

Vom implementa metodele *compatibleChars()* (verifică dacă două caractere sunt compatibile), *compatibleLines()* (verifică dacă două teste sunt compatibile), *mergeChars()* (fuzionarea a două caractere compatibile) şi *mergeLines()* (fuzionează două teste compatibile).

Program

```
#include <fstream>
#include <string>
#include <vector>

using namespace std;

bool compatibleChars(char c1, char c2){
  return(
    c1 == c2    ||
    c1 == 'X'   ||
    c2 == 'X'
    );
}

bool compatibleLines(string str1, string str2){
  if(str1.length()!=str2.length()) return false;
  bool retValue = true;
  for(int i=0; retValue && i<str1.length(); i++)
    retValue &= compatibleChars( str1[i], str2[i] );
  return retValue;
}

char mergeChars(char c1, char c2){
  if(!compatibleChars(c1, c2)) return '*';
  char retCh;
  if('X'==c1 && 'X'==c2){
    retCh = 'X';
    } else{
      if(c1!='X') retCh = c1;
```

Două caractere sunt compatibile atunci când coincid sau când unul dintre ele este X (*don't care*).

```
      else retCh = c2;
    }
  return retCh;
}

string mergeLines(string str1, string str2){
  if(str1.length() != str2.length()) return NULL;
  char* str =(char*) malloc(str1.length() + 1);
  for(int i=0; i<str1.length(); i++)
    str[i] = mergeChars(str1[i], str2[i]);
  str[str1.length()] = '\0';
  return str;
}

void recExactCompact(vector<string> v, vector<string>& vOut){
  bool compact = true;
  int i, j;
  string str;
  for(i=0; i<v.size()-1; i++)
    for(j=i+1; j<v.size(); j++)
      if(compatibleLines(v[i], v[j])){
        compact = false;
        str = mergeLines(v[i], v[j]);
        v.erase(v.begin()+i);
        v.erase(v.begin()+j-1);
        v.push_back(str);
        recExactCompact(v, vOut);
      }
      if(compact){
        if(vOut.size()>v.size()){
          vOut.clear();
          for(int i=0; i<v.size(); i++) vOut.push_back(v[i]);
        }
        return;
      }
}

double doRecExactCompact(string fIn, string fOut){
  ifstream in(fIn.c_str());
  ofstream out(fOut.c_str());
  string line;
  vector<string> v;
  int i, j;
  while(!in.eof()){
    getline(in, line);
    if(line.size()>0)v.push_back(line);
  }
  vector<string> vRet;
```

v nu este compact atunci când există două teste (linii) compatibile! Testul fuzionat va înlocui cele două teste găsite compatibile şi metoda va fi apelată recursiv. Atunci când nu există teste compatibile, vectorul este compact! Mai verificăm în acest caz şi dacă dimensiunea vectorului *v* este mai mică decât cea a lui *vOut*, caz în care acesta din urmă se actualizează.

```
  for(i=0; i<v.size(); i++) vRet.push_back(v[i]);
  recExactCompact(v, vRet);
  out << v.size() << " " << v[0].size()
      << endl << vRet.size() << " "
      << vRet.size()*100.0/v.size() << "%" << endl;
  out << endl;
  for(int i=0; i<vRet.size(); i++){
    out << vRet[i] << endl;
  }
  return vRet.size()*100.0/v.size();
}

int main(){
  doRecExactCompact("tests.in", "tests.out");
  return 0;
}
```

Probleme propuse

1. Scrieți următoarele metode:
   a) Calculează numărul total de perechi compatibile în fișierul de intrare. Exemplu: pentru *test.in* din descrierea problemei, acest număr este 41.
   b) Calculează proprietățile datelor dintr-un fișier de intrare dat: număr de teste, lungimea lor, distribuția caracterelor pe fiecare coloană.
   c) Generează fișiere de intrare cu dimensiunile și valoarea *compaction_factor* (gradul compactării dintre fișierul de intrare și rezultatul așteptat; acesta trebuie să fie aproximat). Exemple pentru gradul de compactare *compaction_factor* 8:

5 linii (teste), 5 coloane (lungimea)	**15 linii (teste), 9 coloane (lungimea)**
1XUU1	XXXXZXXXX
UX1U0	X1X1ZXXXX
Z10XX	X1X1ZXX1X
000X0	X1X1Z111Z
0XZ0U	XXXXZX1XX
	XXXXZXXXX
	11X1Z111Z
	X1X1ZXX1X
	XXX1ZXX1X
	1000U0UU0
	XXXXZXXXX
	X00ZU0ZZU
	XXXXZ11XX
	XXXXZXXXX
	X1X1ZXXXX

2. Din cauza complexității sale exponențiale, algoritmul de tip *Backtracking* recursiv descris mai sus conduce la un timp de execuție foarte ridicat. Din acest motiv, el poate funcționa doar pentru instanțe ale problemei de dimensiune redusă. În scopul compactării unei mulțimi de teste, se poate utiliza un algoritm de tip *Greedy*, care însă nu va furniza mereu cea mai mică mulțime compatibilă. Pseudocodul pentru un astfel de algoritm este următorul:

```
ALGORITHM_GREEDY_COMPACT
        vector v[1..n]
        bool compact
        compact ← false
        While (NOT compact) Execute
                If( ∃ti, tj: ti compatible tj ) Then
                  (consider first (i, j) lexicographical)
                        compact ← false
                        t ← merge(ti, tj)
                        v.delete(ti)
                        v.delete(tj)
                        v.add(t)
                End_If
                Else Execute
                        compact ← true
                End_Else
        End_While
END_ALGORITHM_ GREEDY_COMPACT
```

Acest algoritm este capabil să prelucreze fișiere de intrare de dimensiune mult mai mare.

a) Scrieți o metodă care implementează algoritmul *Greedy*.
b) Adăugați programului de mai sus metodele de la problema anterioară și cu algoritmul *Greedy*.
c) Testați toate aceste metode prin crearea aleatorie de fișiere de intrare și prin compactarea acestora utilizând *Greedy-Compact*. Determinați automat și timpul de execuție pentru metoda *Greedy*. De exemplu, puteți crea automat fișiere de intrare cu următoarele condiții:
   - *compaction_factor* 25;
   - număr de teste de la 100 până la 1000, cu un pas 100 (*f100_xxx.txt* până la *f1000_xxx.txt*);
   - variați și numărul de coloane (lungimea unui test) de la 115 la 915, cu un pas de 200 (*fxxx_115.txt* până la *fxxx_915.txt*).

Fișierul de ieșire *report.out* poate arăta astfel:

report.out

file_name	#lines	#cols	#comps	comp_rate	time(sec)
f100_115.txt	**100**	**115**	**549**	**36.00%**	**1**
f100_315.txt	100	315	850	24.00%	1
f100_515.txt	100	515	1503	20.00%	1
f100_715.txt	100	715	647	28.00%	1
f100_915.txt	100	915	973	26.00%	2
...					
f600_115.txt	**600**	**115**	**35915**	**33.83%**	**82**
f600_315.txt	600	315	29436	30.33%	123
f600_515.txt	600	515	17916	29.83%	163
f600_715.txt	600	715	37335	30.17%	190
f600_915.txt	600	915	8452	26.83%	186
...					
f1000_115.txt	**1000**	**115**	**16218**	**32.80%**	**213**
f1000_315.txt	1000	315	89951	33.90%	394
f1000_515.txt	1000	515	28579	29.70%	813
f1000_715.txt	1000	715	258987	17.90%	227
f1000_915.txt	1000	915	86154	16.30%	495

(#comps *este numărul de perechi compatibile inițiale*)

3. Comparați metodele *Backtracking* recursiv și *Greedy* prin generarea unor fișiere de intrare de diferite dimensiuni și compactarea lor folosind cele două metode. Automatizați procesul, astfel încât să genereze fișiere cu diferite caracteristici, să le compacteze folosind cele două metode, iar rezultatele să fie scrise în fișierul *report.out*. Caracteristicile pot să fie, de exemplu, *compaction_factor* 50; număr de teste de la 5 la 35, fiecare cu număr de coloane de la 5 la 9 (amândouă cu pasul 1) și rezultatele trebuie scrise ca mai jos. Marcați cu ** cazurile când algoritmul optim de tip *Backtracking* furnizează o soluție mai bună decât *Greedy*. Alternați factorul de compactare *compaction_factor* pentru fișierele de intrare, numărul de teste și lungimea acestora. Când furnizează algoritmul *Backtracking* rezultate mai bune decât cel *Greedy*?

report.out

				GREEDY_COMPACT		BACK_COMPACT		
file_name	#lin	#col	#cps	c_rate	t(s)	c_rate	t (s)	
f_5_5.txt	**5**	**5**	**3**	**60.00%**	**0**	**60.00%**	**0**	
f_5_6.txt	5	6	1	80.00%	0	80.00%	0	
f_5_7.txt	5	7	4	60.00%	0	60.00%	0	
...								
f_11_7.txt	11	7	12	54.55%	0	45.45%	0	**
f_11_8.txt	11	8	13	45.45%	0	45.45%	0	
f_11_9.txt	11	9	4	63.64%	0	63.64%	0	
f_12_5.txt	**12**	**5**	**58**	**16.67%**	**0**	**16.67%**	**0**	
...								

f_20_6.txt	20	6	48	55.00%	0	55.00%	0	
f_20_7.txt	20	7	31	35.00%	0	35.00%	3	
f_20_8.txt	20	8	61	50.00%	0	50.00%	1	
...								
f_35_5.txt	**35**	**5**	**90**	**54.29%**	**0**	**45.71%**	**395**	**
f_35_6.txt	35	6	201	40.00%	0	37.14%	3689	**
f_35_7.txt	35	7	150	40.00%	0	40.00%	2677	
f_35_8.txt	35	8	71	57.14%	0	57.14%	82	
f_35_9.txt	35	9	116	51.43%	0	51.43%	254	

4. Testele sunt în program de tip *string*, deşi mulţimea de compactare {0, 1, U, X, Z} are doar cinci elemente (aceasta înseamnă că sunt folosiţi 8 biţi pentru un caracter, când, de fapt, ar fi suficienţi trei). Îmbunătăţiţi programul astfel încât reprezentarea testelor să fie mai economică, prin folosirea operatorilor pe biţi, de exemplu (sau a tipurilor *bitset*, *vector<bool>* din *STL*).
5. Dezvoltaţi programul astfel încât să nu se parcurgă întotdeauna întregul vector pentru găsirea perechilor compatibile de teste. Acestea trebuie doar salvate într-o listă şi cu ajutorul ei trebuie prelucrate.

Încă 12 probleme propuse

1. *Parantetizarea corectă a* n *perechi de paranteze.* Scrieţi un program care să furnizeze toate parantetizările corecte ale *n* perechi de paranteze. Exemplu:

Tastatură	paranteze.out
n=3	()()() , (()()), (())(), (()()), ((()))

2. *Colorarea hărţii.* Se dă o hartă cu *n* ţări ($2 \le n \le 20$) sub forma unei matrici *a*[][], în care *a*[*i*][*j*] = 1, dacă ţările *i* şi *j* sunt vecine şi *a*[*i*][*j*] = 0, dacă nu sunt vecine. Determinaţi o modalitate de a colora harta cu un număr minim de culori, astfel încât oricare două ţări vecine să fie colorate diferit. Scrieţi pe prima linie a fişierului de ieşire *culori.out* numărul de culori găsit, iar pe următoarea linie lista culorilor corespunzătoare, începând cu ţara 1 până la ţara *n*.

harta.in	culori.out
7 0 1 1 1 0 0 1 1 0 1 1 0 0 0 1 1 0 1 1 0 1 1 1 1 0 1 0 1 0 0 1 1 0 1 1 0 0 0 0 1 0 1 1 0 1 1 1 1 0	4 1 2 3 4 1 3 2

Generaţi toate posibilităţile de a colora harta cu numărul minim de culori.

3. *Produs cartezian.* Produsul cartezian a n mulțimi $A_1, A_2, ..., A_n$ este format din toate n-uplele $(a_1, a_2, ..., a_n)$ cu $a_i \in A_i$. Acesta se notează cu $A_1 \times A_2 \times ... \times A_n$:

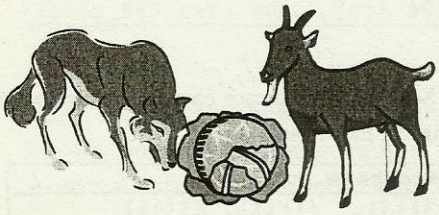
$$\prod_{i=1}^{n} A_i = A_1 \times A_2 \times ... \times A_n = \{(a_1, a_2, ..., a_n) | a_i \in A_i \ \forall i = 1,2, ..., n\}$$

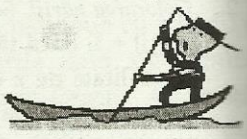
Numărul total al n-uplelor este $|A_1| \cdot |A_2| \cdot ... \cdot |A_n|$.

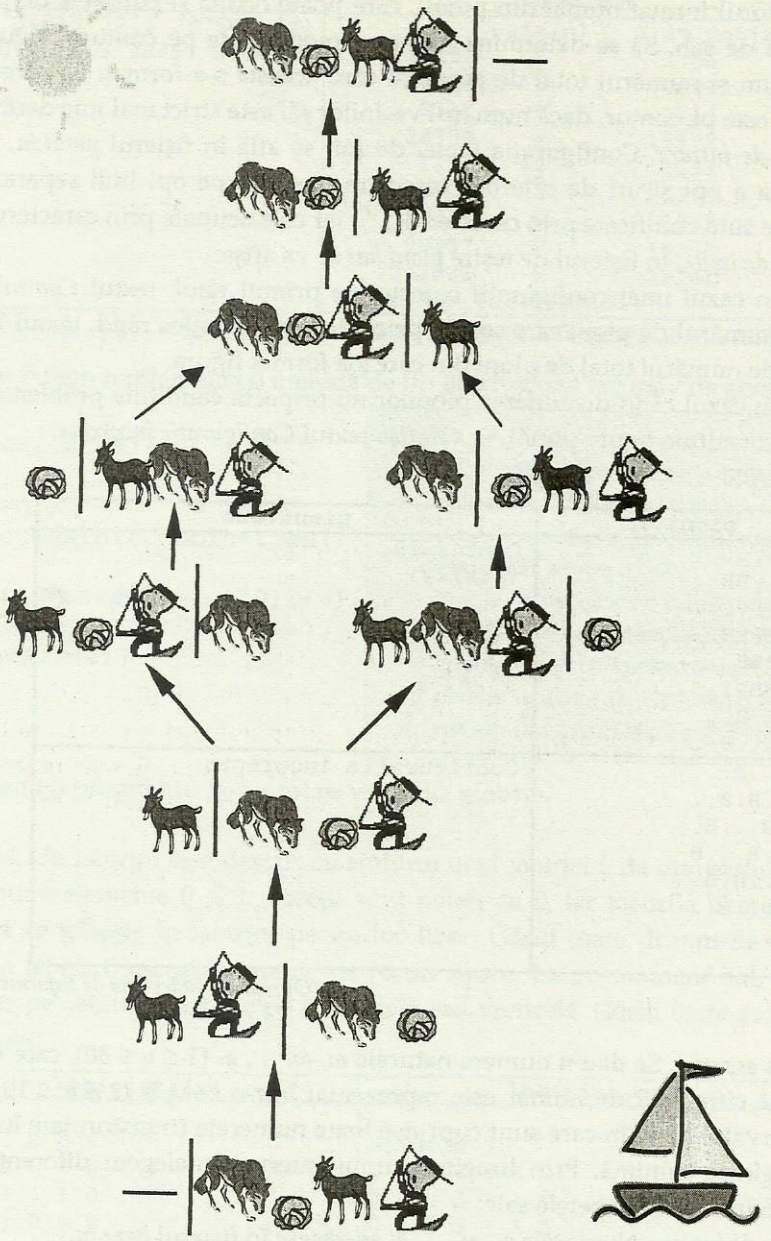
În această problemă, ne dorim determinarea produsului cartezian {1, 2, ..., n_1} × {1, 2, ..., n_2} × ... × {1, 2, ..., n_k}. În fișierul *cart.in* se găsesc dimensiunile mulțimilor. Scrieți toate k-uplele în fișierul *cart.out*, câte unul pe linie. Exemplu:

cart.in	cart.out
2 3 1 4	1 1 1 1 1 1 1 2 ... 2 3 1 4

4. *Ce face un țăran cu un lup?* Un țăran se află la drum cu o capră, o varză și un lup și dorește să treacă împreună cu toți însoțitorii un râu. El găsește pe mal o barcă; aceasta este însă atât de mică, încât cu mare greutate încape el și unul dintre însoțitori. Deoarece numai doi încap în barcă și el vrea ca toți să ajungă întregi pe celălalt mal, nu are voie să lase cumva capra cu lupul sau capra cu varza nesupravegheați, căci capra poate fi mâncată de lup sau capra poate mânca varza. Găsiți o reprezentare a problemei și rezolvați dilema țăranului folosind metoda *Backtracking*. Câte posibilități există?







Țăranul, lupul, capra și varza

5. *Pioni pe tabla de șah.* Pe o tablă de șah de dimensiune 8 × 8 se găsesc pioni. Vecinii unui pion se află în câmpurile imediat vecine pe orizontală, verticală și diagonală. Pionii sunt aranjați astfel încât formează o singură figură plină (fără orificii, cu

interiorul format numai din pioni), care poate ocupa şi pătratele de pe marginea tablei de şah. Să se determine numărul pionilor de pe conturul figurii formate, precum şi numărul total de pioni cu care aceasta s-a format. Se consideră că un pion este pe contur, dacă numărul vecinilor săi este strict mai mic decât 8.

Date de intrare. Configuraţia tablei de şah se află în fişierul *pioni.in*. Aceasta are forma a opt şiruri de câte opt caractere, dispuse pe opt linii separate. Pătratele libere sunt codificate prin caracterul „." , iar cele ocupate prin caracterul „*B*" .

Date de ieşire. În fişierul de ieşire *pioni.out* se va afişa:

a) în cazul unei configuraţii corecte: pe primul rând, textul *Contur=*, urmat de numărul de pioni care se află pe contur; pe al doilea rând, textul *Total=* urmat de numărul total de pioni din care s-a format figura.

b) în cazul când dispunerea pionilor nu respectă condiţiile problemei (pionii nu constituie figură plină), se va afişa textul *Configuratie incorecta.*

Exemple:

pioni.in	pioni.out
....BB.. ...B.... .BBBBB.. BBBBB... BBBBB... ..B.B... .....B..	Contur=19 Total=21
....B... ...B.B.. ..B...B. ...B...B ....B.B. .....B..	Configuratie incorecta

(Olimpiada Judeţeană de Informatică, Iaşi, 2000)

6. *Baze ascunse.* Se dau n numere naturale $a_1, a_2, \ldots, a_n$ ($1 \le n \le 30$), care au maximum nouă cifre. Fiecare număr este reprezentat într-o bază b_i ($2 \le b_i \le 10$), astfel încât intervalul $[a, b]$ în care sunt cuprinse toate numerele (transformate în baza 10) are lungimea minimă. Prin lungimea unui interval înţelegem diferenţa în valoare absolută dintre capetele sale.

Date de intrare. Numerele $a_1, a_2, \ldots, a_n$ se găsesc în fişierul *baze.in*.

Date de ieşire. Scrieţi în fişierul *baze.out* perechile (a_i, b_i) şi pentru fiecare reprezentarea în baza 10, ca în exemplu. Pe ultima linie scrieţi şi intervalul în care se găsesc aceste numere. Exemplu:

baze.in	baze.out
12102 34215 2314 28756 1231 1010101 23413 28457 343421	n Baza(n) n in zecimal 12102 6 1766 34215 6 4835 2314 9 1714 28756 9 19572 1231 10 1231 1010101 4 4369 23413 5 1733 28457 9 19330 343421 5 12361 Interval: [1231,19572]

(Olimpiada Națională de Informatică, Timișoara, 1997, enunț modificat)

Indicație: Putem implementa o metodă de tip *Backtracking* recursiv de genul:

```
void back(int k){
  int i;
  if(k==n)
     processThisCombination();
   else
     for(i=base[k];i<=10;i++){
      x[k]=i;
      back(k+1);
      }
}
```

Dacă:
- a fost găsită o configurație de baze ($k = n$), atunci verificăm dacă intervalul corespunzător este mai scurt decât cel mai bun găsit până acum;
- încă nu a fost găsită o configurație de baze ($k < n$), atunci se setează $x[k]$ cu toate bazele posibile și se apelează metoda pentru nivelul următor ($k + 1$) ($base[k]$ este valoarea minimă posibilă a bazei pentru a_k).

Implementați programul fără a folosi variabile globale.

7. *Labirintul.* Un labirint este descris cu ajutorul unei matrici L de dimensiune $m \times n$, care conține elemente 0 și 1. Pereții sunt notați cu 0, iar locurile libere cu 1. O persoană se găsește în labirint pe un loc liber. Găsiți toate drumurile care duc afară din labirint, mergând numai pe locuri libere. La un moment dat, se poate păși doar pe celula vecină de pe orizontală sau verticală. Găsiți toate soluțiile, ca în exemplu:

labirint.in	labirint.out
4 7 0 0 1 1 0 0 0 0 0 0 1 0 1 0 0 0 1 1 1 1 0 0 1 1 1 0 0 0 3 3	... Solutia 3: 0 0 1 **4** 0 0 0 0 0 0 **3** 0 1 0 0 0 **1** **2** 1 1 0 0 1 1 1 0 0 0 ...

8. *Steagurile.* Ne imaginăm că avem la dispoziție stofe de trei culori: alb, galben, portocaliu, roșu, albastru, verde și negru. Se cere determinarea tuturor

posibilităților de a crea steaguri cu trei culori pe orizontală, astfel încât culorile să fie diferite și în mijloc să fie alb, galben sau portocaliu. Scrieți toate posibilitățile în fișierul *steaguri.out*, câte una pe linie:

steaguri.out
galben alb portocaliu ... albastru galben rosu ... negru portocaliu verde

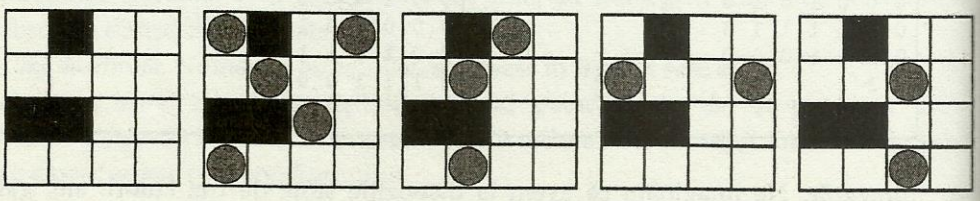
Fără program, doar pe hârtie, câte steaguri se pot crea?

9. *Câte resturi poți să îmi dai?* Găsiți toate posibilitățile de a plăti suma S dată cu monede diferite de valori $m_1, m_2, ..., m_n$. În fișierul *plata.in* se găsește suma S pe prima linie, urmată de valorile monedelor ($0 \leq S \leq 1200$, $1 \leq m_i \leq 50$). Toate modalitățile se vor scrie în fișierul *plata.out*. Exemplu:

plata.in	plata.out
136 12 1 5 25	Solutie: 4x2 + 11x12 ... Solutie: 1x1 + 2x5 + 5x25

10. *Ecuație cu trei necunoscute.* Scrieți în fișierul *ecuatie.out* toate tripletele $\{x, y, z \in \mathbb{N} \mid 2xy + xyz + 4z = 152\}$. Generalizați problema.

11. *Ture pe o altfel de tablă de șah.* În șah, tura este piesa care poate fi mutată oricâte pătrate vertical și oricâte pătrate orizontal. În această problemă vom considera o tablă de șah de dimensiune mică (cel mult 4 × 4) ce poate conține și ziduri peste care nu se poate trece. Scopul este plasarea, cât mai multor ture pe tablă astfel încât să nu se poată captura una pe alta. O configurație de ture se consideră *corectă* dacă două ture nu sunt pe același rând orizontal sau vertical sau, dacă sunt, ele trebuie să fie separate printr-un zid. Următoarea imagine prezintă cinci stări ale unei table de șah.



Prima este o tablă fără ture, a doua şi a treia reprezintă configuraţii corecte, a patra şi a cincea reprezintă configuraţii incorecte. Pentru această tablă, numărul maxim de ture ce pot fi amplasate este 5, iar figura 2 este o modalitate de amplasare.

Date de intrare. În fişierul *ture.in* se găsesc mai multe configuraţii, în forma următoare: pe o linie, un număr natural n ($1 \leq n \leq 4$) reprezentând dimensiunea tablei; pe următoarele n linii, tabla de şah exprimată prin caractere.

Date de ieşire. În fişierul *ture.out* trebuie să se scrie, pentru fiecare set de date din fişierul de intrare, numărul maxim de ture ce pot fi amplasate pe o tablă corespunzătoare. Exemplu:

ture.in	ture.out
1 . 2 XX XX 2 X. X. 3 ... ... 4 .... ..X. .X.. .... 4 XXX. .... .X.. XXX. 3 .X. ..X .X.	1 0 1 3 6 4 4

(ACM, South-Central USA Collegiate Programming, 1998)

12. *Tanţa florăreasa.* Doamna Tanţa este o florăreasă grăbită, iar în unele zile se încurcă cu florile sale şi nu ştie cum să rezolve aşezarea lor în cutii. Dimineaţa, toate florile sunt amestecate în cele, să zicem, n cutii. Tanţa are n categorii de flori şi ar dori să le sorteze, fiecare tip de floare în cutia sa, printr-un număr minim de mutări. Ea îşi iubeşte florile şi le mută cu mare grijă, câte una; deci, prin *mutare* înţelegem mutarea unei singure flori în altă cutie. Va trebui să-i daţi o mână de ajutor, căci

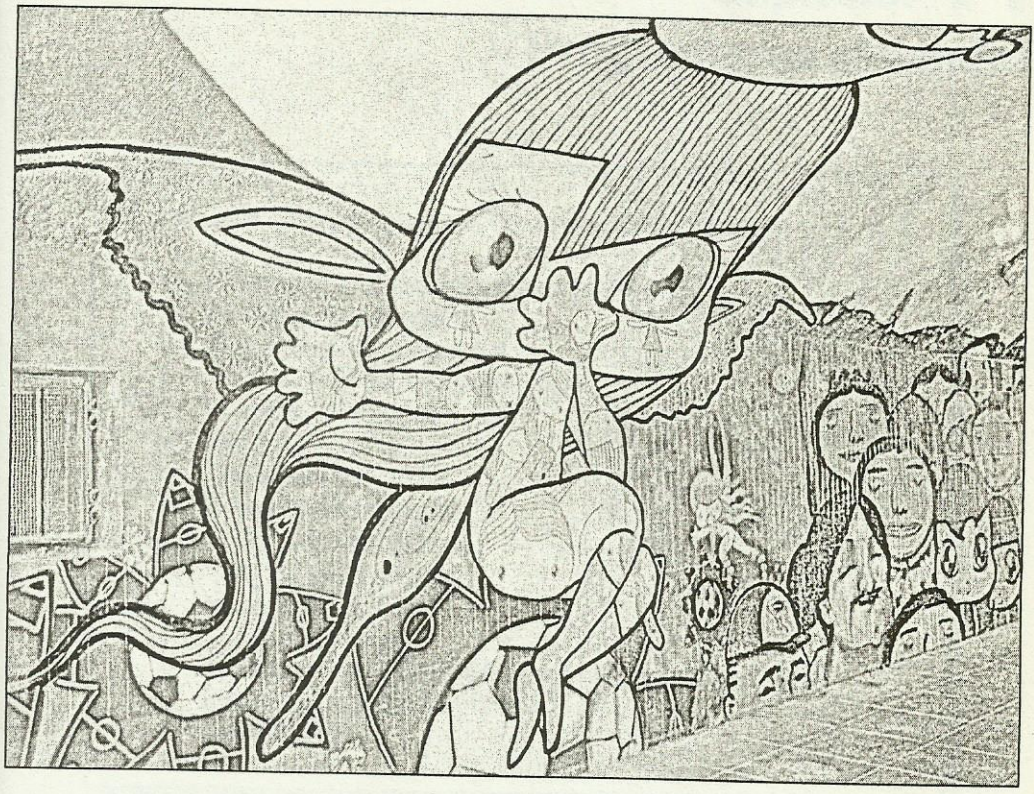
doamna Tanța este o persoană bună și vă va răsplăti cu un buchet minunat. Doamnei Tanța nu-i place să aibă flori cu aceleași inițiale, așa că florile sale încep cu litere diferite. Ea are cele mai frumoase flori, dar niciodată mai mult de 10 varietăți. Presupunem că stările inițiale ale cutiilor se află în fișierul *flori.in*. Pentru fiecare instanță, avem pe prima linie numărul n de cutii și de soiuri de flori, pe a doua linie, șirul reprezentând inițialele florilor așa cum sunt ele considerate la început în fiecare cutie. Apoi, urmează un șir cu $N \times N$ elemente. Primele N elemente reprezintă așezarea florilor în prima cutie, următoarele N elemente reprezintă așezarea florilor în a doua cutie etc. Datele de intrare se termină cu numărul 0. Va trebui să scrieți un program care memorează în fișierul *flori.out* un raport referitor la aceste instanțe; pentru fiecare dintre ele se vor scrie: numărul cazului analizat, numărul minim de mutări, ce fel de tip de floare și câte din fiecare se află în final în fiecare cutie, ca în exemplul următor.

Observație:

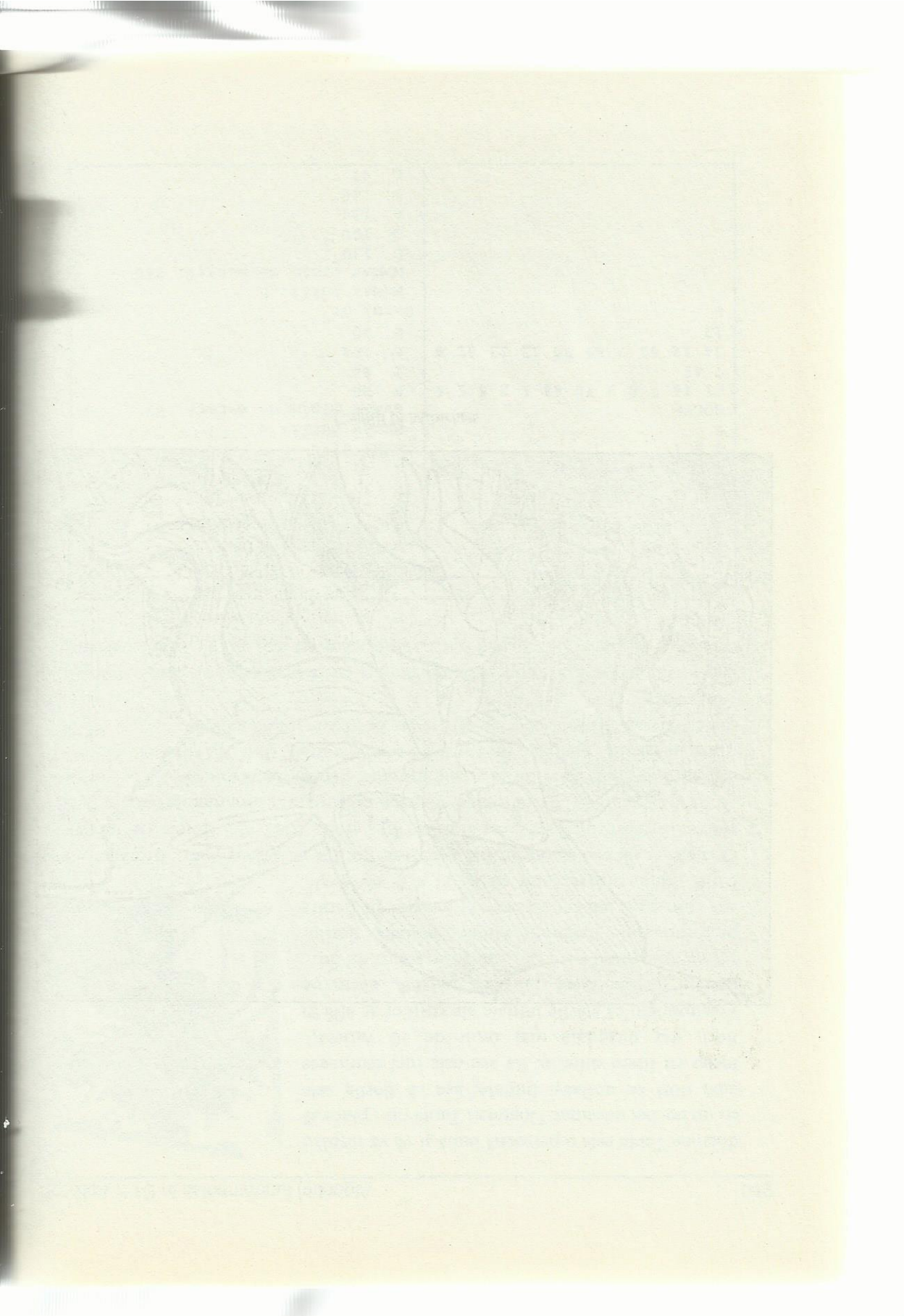
Dacă sunt mai multe modalități de mutări cu obținerea unui număr minim, se consideră numai acela cu șirul inițialelor florilor cel mai mic în ordinea alfabetică.

Exemplu:

flori.in	flori.out
3 BGC 5 10 5 20 10 5 10 20 10 4 TCDF 4 6 7 8 9 1 2 4 3 9 90 7 8 4 5 1 5 GOLTA 21 14 1 8 9 70 43 1 2 3 5 4 7 43 34 12 65 7 89 76 12 13 31 4 12 0	Cazul 1: Numar cutii: 3 Numar minim de mutari: 50 C 20 B 35 G 40 Cazul 2: Numar cutii: 4 Numar minim de mutari: 57 F 20 T 24 D 104 C 20 Cazul 3: Numar cutii: 5 Numar minim de mutari: 348 O 139 G 120 A 134 T 146 L 47



Graffiti în München



Capitolul 11
Programarea dinamică

14 probleme complet rezolvate, 54 de probleme propuse

- Fundamente şi proprietăţi ale metodei
- Numărarea iepuraşilor
- Subşir crescător maximal
- Cel mai lung subşir comun (*LCS*)
- Triunghi de numere
- Domino
- Împărţirea cadourilor
- Sume asemănătoare
- Scoţieni la Oktoberfest
- Calul pe tabla de şah
- Fluctuaţii la bursă
- Sume de produse
- Triangularizare minimală a unui poligon convex
- Înmulţirea unui şir de matrice
- *Edit-Distance*

Magia se ascunde întotdeauna în detalii.
Theodor Fontane

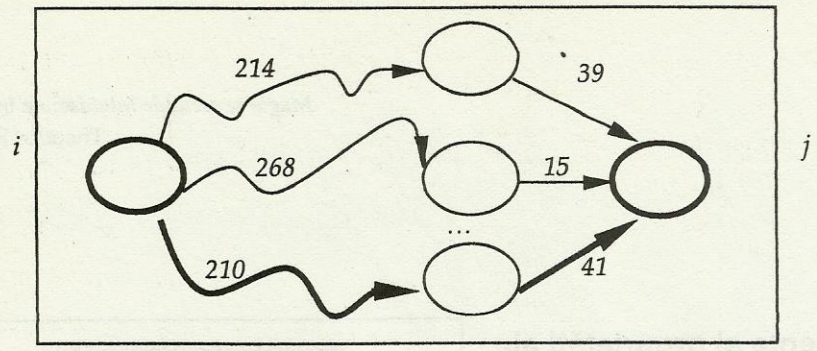
Fundamente şi proprietăţi ale metodei

Richard Bellman
®IEEE History Center,
1979 IEEE Awards Reception Brochure
http://www.ieee.org/organizations/history_center/legacies/bellman.html

1. *Apariţia conceptului.* Programarea dinamică este o metodă de programare care rezolvă multe probleme de optimizare. Termenul a fost introdus în 1940 de către matematicianul american Richard Bellman (1920--1984). La început a fost folosit în teoria controlului şi în acest domeniu se vorbeşte despre principiul lui Bellman al programării dinamice.

2. *Principiul optimalităţii.* Metoda programării dinamice se bazează pe principiul optimalităţii. Aceasta înseamnă că problema dată se poate împărţi în subprobleme asemănătoare de dimensiuni mai mici, iar unele dintre soluţiile acestor subprobleme participă la determinarea soluţiei optime pentru problema dată. Principiul optimalităţii este caracteristic şi subproblemelor.

Să ne imaginăm că dorim să determinăm drumul de cost minim dintre două vârfuri date i şi j ale unui graf. Pentru aceasta, calculăm drumurile minime de la i la toate vârfurile vecine lui j. Pe baza acestora şi pe baza costurilor muchiilor de la vecini la vârful j, determinăm drumul minim de la i la j. Determinarea lungimii drumului de la i la vecinii lui j sunt subprobleme ale problemei date.



Determinarea drumului minim dintre două vârfuri ale unui graf

De obicei, soluția unei probleme asupra căreia se aplică principiul optimalității este bazată pe trei pași:

1. împărțirea problemei în probleme de dimensiuni mai mici;
2. rezolvarea optimală a subproblemelor tot pe baza acestui model în trei pași;
3. combinarea soluțiilor subproblemelor pentru găsirea soluției optimale a problemei date.

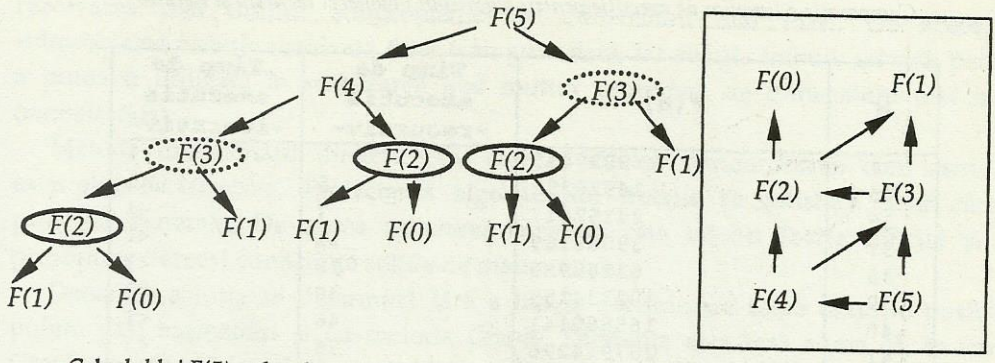
3. *Suprapunerea problemelor și salvarea soluțiilor optimale ale subproblemelor* (memoization). Alte două caracteristici ale metodei sunt suprapunerea și salvarea soluțiilor optimale ale subproblemelor. *Suprapunerea* înseamnă că soluția optimală a unei subprobleme poate să contribuie la determinarea soluției optime a mai multor probleme de dimensiuni mai mari. *Memoization* înseamnă că soluțiile subproblemelor sunt salvate pentru a fi folosite ulterior, ceea ce îmbunătățește semnificativ timpul de execuție.

4. *Exemplu introductiv: șirul lui Fibonacci.* Considerăm problema calculării termenilor din șirul lui Fibonacci, care este definit astfel:

- $F(0) = 0, F(1) = 1$
- $F(n) = F(n-1) + F(n-2)$ pentru toți $n \geq 2$ (1)

Pentru a calcula $F(5)$, avem nevoie de numerele $F(4)$ și $F(3)$.

În general, pentru a putea calcula $F(n)$ avem nevoie de $F(n-1)$ și $F(n-2)$. Dacă aceste numere ar fi calculate independent – de exemplu, cu un algoritm recursiv –, ar trebui ca același număr să fie determinat de mai multe ori. Considerăm $F(5)$:



Calculul lui F(5): arborele recursiv şi digraful aciclic (engl., Directed acyclic graph, DAG*)*

În cazul recursiv, am calcula $F(2)$ de trei ori şi $F(3)$ de două ori, fiecare independent. O altă metodă ar fi calcularea valorilor $F(2)$, $F(3)$, $F(4)$, ..., $F(n)$ şi salvarea rezultatelor. Formal, cele două variante se pot scrie:

Algoritmii recursiv şi iterativ pentru calcularea numerelor din şirul lui Fibonacci

Determinarea recursivă a celui de-al n-lea termen Fibonacci	Determinarea iterativă a celui de-al n-lea termen Fibonacci (programare dinamică)
Fib(n) **If**($n == 0$ **or** $n == 1$) **Then** return n **End_If** **Else Execute** return Fib($n-1$) + Fib($n-2$) **End_Else** ***End_Fib(n)***	***Fib(n)*** $F[0] \leftarrow 0$ $F[1] \leftarrow 1$ **For**($i = 2$; $i \leq n$; step 1) **Execute** $F[n] \leftarrow F[n-1] + F[n-2]$ **End_For** return $F[n]$ ***End_Fib(n)***

Folosim un program pentru a compara ambii algoritmi, care afişează şi timpii de execuţie pentru fiecare metodă în parte.

Comparație a timpilor de execuție pentru algoritmii Fibonacci recursiv și iterativ

N	F(n)	Timp de executie -recursiv-	Timp de executie -iterativ-
34	9227465	3	<1
35	14930352	5	<1
36	24157817	7	<1
37	39088169	11	<1
38	63245986	18	<1
39	102334155	30	<1
40	165580141	46	<1
41	267914296	76	<1
42	433494437	122	<1
43	701408733	198	<1
44	1134903170	320	<1
45	1836311903	522	<1
46	2971215073	845	<1
47	4807526976	1361	<1
48	7778742049	2211	<1
49	12586269025	3710	<1
50	20365011074	5904	<1

În această tabelă, se remarcă faptul că timpul de execuție pentru varianta recursivă crește extrem de repede, din cauză că fiecare număr este calculat independent. În acest caz, complexitatea algoritmului este exponențială: $O(2^n)$. A doua variantă folosește principiul optimalității. Fiecare număr este calculat numai o dată și valoarea lui este salvată într-un vector, pentru a putea fi folosită după aceea. De aceea, complexitatea este liniară: $O(n)$.

5. *Bottom-up* vs *top-down*. De obicei, metoda folosește una dintre următoarele strategii:

Bottom-up. Toate subproblemele care ar putea fi folosite ulterior sunt rezolvate dinainte. După aceea, soluțiile lor vor fi folosite pentru a determina soluțiile optimale ale unor subprobleme de dimensiune mai mare. De obicei, se completează o tabelă cu soluțiile subproblemelor și, la sfârșit, soluția problemei inițiale se găsește în aceasta. Avantajul este că manevrarea eficientă a tabelei economisește timp.

Top-down. Problema se împarte în subprobleme de dimensiune mai mică, ce sunt rezolvate, iar soluțiile sunt salvate pentru cazul când ele vor fi necesare. O subproblemă se rezolvă o singură dată.

6. *Comparația cu alte metode de programare.* Altfel decât în cazul metodei *Divide et Impera*, unde subproblemele trebuie să fie independente una de alta, în cazul programării dinamice, poate exista situația când o subproblemă participă la

rezolvarea mai multor subprobleme de dimensiuni mai mari. De aceea, o subproblemă trebuie rezolvată doar o singură dată, iar soluția trebuie salvată, pentru a putea fi utilizată în rezolvarea mai multor probleme de dimensiuni mai mari (*memoization*).

Metoda programării dinamice nu este o tehnică standardizată, cum este, de exemplu, *Backtracking*. Proiectarea algoritmului trebuie să satisfacă doar câteva principii generale. De aceea utilizarea metodei este uneori foarte dificilă şi ea presupune deseori cunoştinţe solide de matematică.

Deoarece soluția se determină fără a lua în considerare toate cazurile posibile, putem găsi asemănări şi cu metoda *Greedy*. Diferența este însă aceea că, în cazul *Greedy*, se adaugă la soluția parțială un optim local, iar soluția nu este, de obicei, cea optimă, pe când programarea dinamică foloseşte principiul optimalității şi conduce la o soluție optimă.

Probleme propuse

1. Definiți conceptele: principiul optimalității, suprapunerea problemelor, *memoization*.
2. Care este procedura de lucru a metodei?
3. Un alt exemplu tipic pentru folosirea metodei este calcularea coeficienților binomiali:

   $C_n^k = 1$, dacă $k = 0$ sau $n = k$

   $C_n^k = C_{n-1}^{k-1} + C_{n-1}^k,\ n > k,\ k \neq 0$ (2)

   Desenați arborele recursiv şi digraful aciclic pentru determinarea valorii C_4^2 (vezi figura pentru calculul lui $F(5)$).

Problema 1. Numărarea iepuraşilor

Leonardo Pisano, cunoscut ca Fibonacci, s-a născut în jurul anului 1180 în Pisa. Deoarece tatăl său făcea comerț cu țările nord-africane, Fibonacci a învățat cifrele hindu-arabe şi metodele de calcul ale matematicienilor arabi. El a scris cartea *Liber Abaci* (*Cartea Abacului*), în care recomandă folosirea cifrelor hindu-arabe şi prezintă probleme matematice spectaculoase, care mai târziu vor fi mereu reluate de către alți autori. De exemplu:

Cineva aşază o pereche de iepuri într-o grădină înconjurată de un zid. Câte perechi de iepuri se vor naşte în fiecare an,

dacă se presupune că, în fiecare lună, fiecare pereche dă naştere unei alte perechi şi că perechile de iepuri sunt fertile la două luni după naştere?

Şirul care se deduce din această problemă, cunoscut ca *şirul lui Fibonacci*, a fost deja prezentat în prima parte a capitolului: $F_0 = 0$, $F_1 = 1$ şi, mai departe, $F_{n+1} = F_n + F_{n-1}$. Aşadar, primii termeni ai şirului sunt: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377.

Scrieți un program care să calculeze cu ajutorul algoritmilor recursiv şi iterativ primii 50 de termeni ai şirului şi să afişeze în plus şi timpii de execuție în secunde pentru fiecare dintre cele două metode:

```
   N       FibRec(N)   Timp(FibRec(N))    FibIt(N)   Timp(FibIt(N))
.........
  33        3524578     t1=      2        3524578     t2=      0
  34        5702887     t1=      3        5702887     t2=      0
  35        9227465     t1=      4        9227465     t2=      0
  36       14930352     t1=      7       14930352     t2=      0
  37       24157817     t1=     11       24157817     t2=      0
.........
```

Analiza problemei şi proiectarea soluției

Vom scrie metodele *fibo1()* şi *fibo2()*, care implementează cei doi algoritmi. Prima metodă determină fiecare termen independent de alți termeni. A doua metodă construieşte iterativ un vector *v* şi, pentru calcularea fiecărei noi valori *v[i]*, foloseşte termenii *v[i–1]* şi *v[i–2]* deja calculați. Pentru determinarea timpului, folosim funcția *time()* din antetul *ctime*. Această funcție salvează timpul într-o variabilă de tip *time_t*, cu dimensiunea de 32 de biți. Astfel, se poate reprezenta timpul în secunde din perioada 01.01.1970, 00:00, până la 18.01.2038, 00:00. Funcția *time()* are sintaxa: *time_t time(time_t *timp_in_secunde)* şi returnează timpul-sistem. Dacă funcția are ca parametru adresa unei variabile de tip *time_t*, atunci timpul se va salva în această variabilă. Dacă se dă ca parametru *NULL*, atunci nu mai are loc această salvare.

Program

```
#include <fstream>
#include <vector>
#include <ctime>

using namespace std;

const int n = 50;

unsigned long long fibo1(int n){
  if(n<=1) return n;
  return fibo1(n-1) + fibo1(n-2);
}

unsigned long long fibo2(int n){
  std::vector<long long> v;
  v.push_back(0);
  v.push_back(1);
  for(int i=2; i<=n; i++ ){
    v.push_back(v[i-1]+v[i-2]);
  }
  return v[n];
}
```

```
int main(){
  unsigned i;
  time_t start, stop;

  ofstream out("FiboReport.txt",
               ios::app);
  for(i=0; i<=n; i++){
    out << endl;
    out.width(4); out<<i;
    start = time(NULL);
    out.width(15);
    out << fibo1(i);
    stop = time(NULL);
    out << "    t1=";
    out.width(6);
    out << stop-start;
    start = time(NULL);
    out.width(15);
    out << fibo2(i);
    stop = time(NULL);
    out << "    t2=";
    out.width(6);
    out << stop-start;
  }
  return 0;
}
```

Probleme propuse

1. Şirul definit prin $L_0 = 2$, $L_1 = 1$ şi $L_n = L_{n-1} + L_{n-2}$ se numeşte şirul lui Lucas. Deseori numerele lui Lucas se pot exprima elegant cu ajutorul unor sume ale numerelor Fibonacci. Modificaţi programul de mai sus astfel încât să calculaţi numerele lui Lucas în locul celor al lui Fibonacci.
2. Aşa cum am specificat, între şirul lui Lucas şi cel al lui Fibonacci există multe relaţii, cum ar fi: $L_{2n} + 2 \times (-1)^{n-1} = 5F_n^2$. Scrieţi un program care verifică această inegalitate pentru $n = 0, 1, 2, ..., 25$ şi listează rezultatele într-un fişier, de exemplu, în forma:

```
   n       L(2n)+2(-1)^(n-1)      5F(n)^2
-------------------------------------------
   0                       0            0  OK!
   1                       5            5  OK!
   2                       5            5  OK!
.........
  22              1568397605   1568397605  OK!
  23              4106118245   4106118245  OK!
  24             10749957120  10749957120  OK!
  25             28143753125  28143753125  OK!
```

3. Şirul definit prin $G_0 = 0$, $G_1 = 1$, $G_2 = 2$ (şi $G_3 = 4$) şi $G_n = G_{n-1} + G_{n-2} + G_{n-3} (+ G_{n-4})$ se numeşte şirul lui Tribonacci (respectiv Quadranacci). În general, au loc pentru termenii de bază $G_i = 2^{i-1}$ pentru $i > 0$ şi $G_0 = 0$. Scrieţi un algoritm care calculează al n-lea termen pentru şirul K-bonacci:

$G_0 = 0, G_1 = 1, ..., G_{k-1} = 2^{k-1}$
$G_n = G_{n-1} + G_{n-2} + ... + G_{n-k}$ pentru $n \geq k$

Scrieți un program care pentru numerele naturale k și n date afișează valoarea G_n, care se încadrează în tipul *unsigned long long*. Folosiți metoda programării dinamice. Exemplu:

kBonacci.in	**kBonacci.out**
3 50	n kBonacci(3, n) 0 0 1 1 2 2 3 3 4 6 ... 47 1424681173049 48 2620397211992 49 4819661885417 50 8864740270458
5 57	n kBonacci(5, n) 0 0 1 1 2 2 3 4 4 8 5 15 ... 55 7280158119874827 56 14312414018268228 57 28137465101354548

Scrieți și o metodă recursivă și comparați timpii de execuție.

4. Un alt exemplu clasic de programare dinamică este calcularea coeficienților binomiali C_n^k, care sunt definiți și mai sus în formula (2). Și de această dată, algoritmul iterativ este mult mai eficient decât algoritmul recursiv. Implementați ambele variante și comparați timpii de execuție pentru diferite valori. Afișați într-o formă corespunzătoare și *triunghiul lui Pascal* până la un nivel n dat. Exemplu pentru $n = 3$:

```
0              1
1          1       1
2      1       2       1
3  1       3       3       1
```

Problema 2. Subşir crescător maximal

Propoziţie (Erdős, Szekeres). Într-un şir $a_1, a_2, \ldots, a_{mn+1}$ cu $mn + 1$ numere reale diferite, există întotdeauna un subşir crescător

$$a_{i_1} < a_{i_2} < \ldots < a_{i_{m+1}} \quad (i_1 < i_2 < \ldots < i_{m+1})$$

de lungime $m + 1$, un subşir descrescător

$$a_{j_1} > a_{j_2} > \ldots > a_{j_{n+1}} \quad (j_1 < j_2 < \ldots < j_{n+1})$$

de lungime $n + 1$ sau amândouă. O demonstraţie a acestei propoziţii se poate construi, de exemplu, folosind metoda reducerii la absurd şi principiul lui Dirichlet. ❑

Se cere determinarea subşirului de lungime maximă într-un şir dat cu numere întregi.

Date de intrare. În fişierul *ascending.in* se găseşte un şir cu numere întregi, maxim 2000 şi fiecare mai mic în modul decât 20.000.

Date de ieşire. Scrieţi încă o dată şirul dat în fişierul *ascending.out*, urmat de subşirul crescător maximal, ca în exemplu:

ascending.in	ascending.out
3 5 76 1 45 2 31 89 90 0 4 15 23 47 95 21 67 8 13 11 5 145 132 77	--- Input data: Length: 24 Elements: 3 5 76 1 45 2 31 89 90 0 4 15 23 47 95 21 67 8 13 11 5 145 132 77 --- Output data: Maximal ascending substring: 1 2 4 15 23 47 95 145 Length: 8

Analiza problemei şi proiectarea soluţiei

Construim vectorul a ce conţine şirul dat în fişierul de intrare şi apoi vectorii *vPred* şi v cu semnificaţiile:

- $v[0] \leftarrow 1$
- $v[i] \leftarrow 1 + \max\{v[j] \mid j < i \text{ şi } a[j] < a[i]\}$, $i = 1, \ldots, n-1$
- $vPred[0] \leftarrow -1$ (primul element nu are nici un predecesor)
  Dacă $\exists$ *jmax* a.î. $v[jmax] = max\{v[j] \mid j < i \text{ şi } a[j] < a[i]\}$, atunci $vPred[i] = jmax$, altfel $vPred[i] = -1$ (nu există nici un predecesor).

$v[i]$ reprezintă lungimea subşirului maximal care îl are pe $a[i]$ ca ultim element.

Subproblemă. Fie şirul $a_1, a_2, ..., a_i$ dat. Determinați cel mai lung subşir al său care îl are pe a_i ca ultim element.

vPred[i] este indexul predecesorului elementului a_i în acest subşir crescător maximal cu a_i ca ultim element. Acest vector ne va ajuta să reconstruim recursiv subşirul cu ajutorul funcției *recoverSubstring()*.

Program

```
#include <fstream>
#include <vector>

using namespace std;

vector<int> vPred;
vector<int> a, v;
int imax;

void readData(ifstream& in){
  int aux;
  while(in && !in.eof()){
    in >> aux;
    a.push_back(aux);
  }
}

void recoverSubstring(int i, ofstream& out){
  if(vPred[i]+1) recoverSubstring(vPred[i], out);
  out << a[i] << " ";
}

void process(){
  int j, i, n;
  v.push_back(1);
  vPred.push_back(-1);
  imax = 0;
  n = (int)a.size();
  for(i=1; i<n; i++){
    v.push_back(1);
    vPred.push_back(-1);
    for(j=0; j<i; j++)
      if(a[j]<a[i]&&v[j]+1>v[i])
      {
        v.pop_back();
        v.push_back(v[j]+1);
        vPred.pop_back();
        vPred.push_back(j);
      }
```

Această buclă *for* poate fi scrisă şi astfel (folosim în blocul *if* operatorul de acces [] în locul funcțiilor specifice vectorului *STL push_back* şi *pop_back*):

```
...
if(a[j]<a[i]&&v[j]+1>v[i]){
  v[i] = v[j] + 1;
  vPred[i] = j;
}
...
```

```
        if(v[i]>v[imax]) imax=i;
    }
}

void writeData(ofstream& out){
    out << "--- Input data: " << endl;
    out << "Length: " << (int)a.size() << endl;
    out << "Elements: ";
    for(int i=0; i<(int)a.size(); i++)
        out << a[i] << " ";
    out << endl;
    out << "--- Output data: " << endl;
    out << " Maximal ascending substring: ";
    recoverSubstring(imax, out);
    out << endl << " Length: " << v[imax] << endl;
}

int main(){
    ifstream in("ascending.in");
    ofstream out("ascending.out");
    readData(in);
    process();
    writeData(out);
    return 0;
}
```

Probleme propuse

1. Dezvoltați programul astfel încât să fie scris și subșirul descrescător maximal în fișierul de ieșire:

substring.in	substring.out
3 5 76 1 45 2 31 89 90 0 4 15 23 47 95 21 67 8 13 11 5 145 132 77 -1 -2 57	--- Input data: Length: 24 Elements: 3 5 76 1 45 2 31 89 90 0 4 15 23 47 95 21 67 8 13 11 5 145 132 77 --- Output data: Maximal ascending substring: 1 2 4 15 23 47 95 145 Length: 8 --- Maximal descending substring: 76 45 31 23 21 13 11 5 -1 -2 Length: 10

2. Se poate să existe mai multe subșiruri cu aceeași lungime maximală. Modificați programul astfel încât să fie afișate toate aceste subșiruri.
3. Scrieți un nou program ce „verifică" propoziția lui Erdős și Szekeres pentru mai multe seturi de date. Acesta trebuie să funcționeze astfel: se generează aleatoriu P numere naturale, pentru orice m, n cu proprietatea că $m \leq n$ și $P = mn + 1$, se determină subșirurile maximale crescător și descrescător. În final, se verifică propoziția.
4. *Cuvinte potrivite.* Într-un fișier se găsește un text scris pe mai multe linii, format din cuvinte ce folosesc doar literele mici ale alfabetului englez. Scrieți un program care determină subșirul maximal de cuvinte din text pentru care este îndeplinită condiția că ultimele două litere ale unui cuvânt din șir coincid cu primele două litere din cuvântul următor.
   Date de intrare. În fișierul *cuvinte.in* se găsesc cel mult 1000 de cuvinte (fiecare de lungime minim 2 și maxim 100), care sunt separate prin spații sau sfârșit de linie (*newline*).
   Date de ieșire. Scrieți pe prima linie a fișierului *cuvinte.out* numărul de cuvinte din subșirul găsit și, începând cu linia următoare, cuvintele în ordine, câte unul pe linie. Exemplu:

cuvinte.in	cuvinte.out
griul visului ultim cazu imprecis in clipa aceea de isterie iesita parca din tainite vechi si tenebroase	7 griul ultim imprecis isterie iesita tainite tenebroase

Problema 3. Cel mai lung subșir comun (*LCS*)

Problema determinării celui mai lung subșir comun (engl., *longest common substring – LCS*) este o problemă clasică de programare dinamică.

Fie $X = (x_1, x_2, ..., x_m)$ și $Y = (y_1, y_2, ..., y_n)$ două șiruri de numere întregi cu n, respectiv m elemente. Se cere determinarea unui subșir maximal comun al celor două date. Exemplu: pentru X = {10, **4**, 20, **10**, 40, **2**, **0**, 60} și Y = {**4**, 90, 7, **10**, 70, **2**, 71, 81, **0**} un cel mai lung subșir comun este Z = {4, 10, 2, 0} și are lungimea 4.

Date de intrare. În fișierul *lcs.in* se află cele două șiruri date, pe prima linie m și n și pe următoarele linii valorile elementelor șirurilor ($1 \leq m, n \leq 500$) de tip *int*.

Date de ieșire. Scrieți un subșir comun maximal în fișierul *lcs.out*, în forma din exemplu.

lcs.in	lcs.out
8 9 10 **4** 20 **10** 40 **2** **0** 60 **4** 90 7 **10** 70 **2** 71 81 **0**	Lungimea maxima: 4 4 10 2 0
14 22 **34** 5 **67** **8** 8 9 **0** **12** **3** **45** 6 78 **91** 231 12 **34** 45 **67** 79 57 **8** 321 55 **0** 33 **12** 1 2 **3** 44 **45** 56 **91** 21 22 23	Lungimea maxima: 8 34 67 8 0 12 3 45 91

Analiza problemei şi proiectarea soluției

Vom nota $X_m = \{x_1, x_2, \ldots, x_m\}$ şi $Y_n = \{y_1, y_2, \ldots, y_n\}$. Vom nota cu $c[i][j]$ lungimea celui mai lung subşir comun al şirurilor $X_i = \{x_1, x_2, \ldots, x_i\}$ şi $Y_j = \{y_1, y_2, \ldots, y_j\}$. Dacă $x_m = y_n$ atunci această lungime este $1 + c[m-1][n-1]$, altfel ea este maximul dintre $c[m-1][n]$ şi $c[m][n-1]$. Pe baza acestei observații, valorile $c[i][j]$ ($0 \le i \le m$, $0 \le j \le n$) se vor calcula secvențial, linie cu linie, de la stânga la dreapta:

```
ALGORITM_LCS (Xm = {x1, x2, ..., xm}, Yn = {y1, y2, ..., yn})
  For (i = 0; i < m; step 1) Execute c[i][0] = 0 End_For
  For (j = 0; j < n; step 1) Execute c[0][j] = 0 End_For

  For (i = 1; i < m; step 1) Execute
    For (j = 1; i < n; step 1) Execute
      If (xi = yj) Then
        c[i][j] = 1 + c[i – 1][j – 1]
       Else c[i][j] = max(c[i – 1][j], c[i][j – 1])
      End_If
    End_For
  End_For
  return c[m][n]
END_ ALGORITM_LCS (Xm = {x1, x2, ..., xm}, Yn = {y1, y2, ..., yn})
```

Pentru primul exemplu, tabelul c[][] creat este redat în figura următoare. Folosim semnele ↑, ←, ← pentru a specifica elementul vecin cu care s-a calculat o valoare $c[i][j]$ (sus, stânga, respectiv diagonală). Drumul pentru construcția subşirului maximal este marcat cu celule gri.

	j	0	1	2	3	4	5	6	7	8	9
i		y_j	4	90	7	10	70	2	71	81	0
0	x_i	0	0	0	0	0	0	0	0	0	0
1	10	0	↑0	↑0	↑0	↖1	←1	←1	←1	←1	←1
2	4	0	←1	←1	←1	↑1	↑1	↑1	↑1	↑1	↑1
3	20	0	↑1	↑1	↑1	↑1	↑1	↑1	↑1	↑1	↑1
4	10	0	↑1	↑1	↑1	↖2	←2	←2	←2	←2	←2
5	40	0	↑1	↑1	↑1	↑2	↑2	↑2	↑2	↑2	↑2
6	2	0	↑1	↑1	↑1	↑2	↑2	↖3	←3	←3	←3
7	0	0	↑1	↑1	↑1	↑2	↑2	↑3	↑3	↑3	↖4
8	60	0	↑1	↑1	↑1	↑2	↑2	↑3	↑3	↑3	↑4

Exemplu pentru algoritmul LCS

După citirea celor două şiruri în vectorii x, respectiv y, vom declara dinamic matricea c cu $m + 1$ linii şi $n + 1$ coloane, pe care o vom iniţializa cu elemente 0:

```
vector< vector<int> > c(m+1, vector<int>(n+1, 0));
```

după care vom aplica algoritmul de mai sus pentru a calcula toate elementele sale iterativ. Pentru a reface subşirul comun maximal, vom folosi c[][], iar valorile comune se vor scrie în vectorul v de la dreapta spre stânga. Se porneşte cu linia m şi coloana n şi se parcurge drumul invers astfel: dacă valoarea X pentru linia curentă ll este egală cu valoarea din Y pentru coloana curentă cc, atunci valoarea comună s-a obţinut printr-o deplasare pe diagonală şi ea se va adăuga lui v, după care ne deplasăm imediat pe diagonală prin decrementarea ll şi cc. Altfel, se procesează analog cazurile pentru care a fost executată o mişcare pe coloană, respectiv pe linie, atunci când a fost calculat elementul $c[ll][cc]$.

Program

```
#include <fstream>
#include <vector>

using std::vector;
using std::ifstream;
using std::ofstream;

int main(){
```

```
ifstream in("lcs.in");
ofstream out("lcs.out");

int m, n;
vector<int> x, y;

if(in && !in.eof() && in>>m>>n){
  int i=0, t;
  for(; in && i<m && in>>t; ++i)
    x.push_back(t);
  for(i=0; in && i<n && in>>t; ++i)
    y.push_back(t);
}

vector< vector<int> > c(m+1, vector<int>(n+1, 0));

for(int i=1; i<=m; ++i)
  for(int j=1; j<=n; ++j){
    if(x[i-1]==y[j-1]){
      c[i][j] = c[i-1][j-1]+1;
      continue;
    };
    if (c[i-1][j]>=c[i][j-1]){
       c[i][j] = c[i-1][j];
    } else {
       c[i][j] = c[i][j-1];
    }
}

int ll=m, cc=n;
vector<int> v;
while(ll || cc){
  if (ll && cc && x[ll-1]==y[cc-1]){
    v.push_back(x[ll-1]);
    --ll; --cc;
    continue;
  }
  if(ll && c[ll][cc]==c[ll-1][cc]){
    ll--; continue;
  }
  if(cc && c[ll][cc]==c[ll][cc-1]){
    cc--;
  }
}

out << "Lungimea maxima: " << c[m][n] << std::endl;
vector<int>::reverse_iterator rIt;
```

```
  for(rIt=v.rbegin(); rIt != v.rend(); ++rIt){
    out << *rIt << "  ";
  }

  return 0;
}
```

Probleme propuse

1. Scrieți o metodă recursivă pentru listarea subșirului comun maximal.
2. *Generalizare.* Pentru un număr natural dat $k \geq 2$ și k șiruri de numere întregi, determinați cel mai lung subșir maximal comun tuturor celor k șiruri.

Problema 4. Triunghi de numere

Considerăm un triunghi de numere ca în exemplul de mai jos și drumurile care duc la numărul de pe prima linie în jos spre bază, astfel încât de fiecare dată se coboară pe linia următoare cu un număr la dreapta celui curent sau imediat dedesubtul acestuia. Găsiți un astfel de drum pentru care suma elementelor constituente să fie maximă.

Date de intrare. În fișierul *triunghi.in* se găsește la început numărul de linii ale triunghiului de numere dat, urmat de acesta, linie cu linie. Numărul maxim de linii ale triunghiului este 100 și fiecare număr este mai mic sau egal cu 100.

Date de ieșire. Scrieți pe prima linie a fișierului *triunghi.out* suma maximală găsită și pe următoarele linii pozițiile din triunghi ce descriu drumul unei soluții cu suma maximală. Exemplu:

triunghi.in	**triunghi.out**
7	Suma maxima = 346
10	1
82 **81**	2
4 6 **10**	3
2 14 **35** 7	3
41 3 **52** 26 15	3
32 90 11 **87** 56 23	4
54 65 89 32 **71** 9 31	5

(IOI, Schweden, 1994)

Analiza problemei și proiectarea soluției

Alegerea la fiecare pas al celui mai mare dintre succesorii posibili pe linia următoare nu conduce de fiecare dată la soluția optimă (aceasta ar fi o soluție de tip *Greedy*). În exemplul nostru, dacă alegem de pe a doua linie a triunghiului 82 în loc de 81, atunci toate sumele care se pot obține mai departe sunt mai mici decât 346 (care este soluția optimală).

Conținutul triunghiului se va citi în tabloul *a*[][] şi construim tabloul bidimensional *s*[][] cu semnificația:

- *s*[*i*][*j*] este cea mai mare sumă până la linia *i* inclusiv, care conține elementul *a*[*i*][*j*] ce se poate obține.

Pentru a salva drumul corespunzător lui *s*[*i*][*j*], folosim tabloul *vPred*[100][100], cu semnificația:

- *vPred*[*i*][*j*] = –1 pentru primul element;
- *vPred*[*i*][*j*] = 0, dacă suma optimală care se termină cu *a*[*i*][*j*] îl conține şi pe *a*[*i* – 1][*j* – 1] (aceasta înseamnă că s-a executat ultima dată un pas spre dreapta-jos);
- *vPred*[*i*][*j*] = 1, dacă suma optimală care se termină cu *a*[*i*][*j*] conține şi pe *a*[*i*–1][*j*] (aceasta înseamnă că s-a executat ultima dată un pas spre direct-jos);
- *vPred*[*i*][0] este mereu 1, deoarece acest element poate fi ultimul al unei sume doar prin executarea unui pas direct în jos (nu există nici un predecesor) *vPred*[*i*–1][–1]!).

Drumul este ulterior salvat în tabloul *w*[], *w*[*i*] este poziția numărului al *i*-lea pe acest drum optimal. După ce tablourile *s*[][] şi *vPred*[] au fost construite, suma optimală se va găsi pe linia *n* a tabloului bidimensional *s*[][]. Mai întâi vom găsi această poziție:

```
w[n-1] = 0;
for(j=1; j<=n-1; j++)
  if(s[n-1][j] > s[n-1][w[n-1]]) w[n-1]= j;
```

Apoi vom utiliza *vPred*[][] pentru a determina toate numerele necesare din triunghi:

```
for(i=n-2; i>=0; i--){
  if(vPred[i+1][w[i+1]] == 0) w[i] = w[i+1]-1;
                                        // pas dreapta-jos
  else w[i] = w[i+1];                   // pas direct-jos
}
```

Program

```
#include <fstream>

using namespace std;

void readData(int a[][100], int &n){
  int i, j;
  ifstream in("triunghi.in");
  in >> n;
  for(i=0; i<n; i++)
    for(j=0; j<=i; j++) in >> a[i][j];
}
```

```
void doProcess(int a[][100], int n, int s[][100],
       int vPred[][100], int w[]){
  int i, j;
  s[0][0] = a[0][0];
  vPred[0][0] = -1;
  for(i=1; i<n; i++){
    s[i][0] = a[i][0] + s[i-1][0];
    vPred[i][0] = 1;
    for(j=1; j<=i; j++){
      if(s[i-1][j-1] >= s[i-1][j]){
       vPred[i][j] = 0;
       s[i][j] = a[i][j]+s[i-1][j-1];
      }else{
        vPred[i][i] = 1;
        s[i][j] = a[i][j]+s[i-1][j];
      }
    }
  }
  w[n-1] = 0;
  for(j=0; j<=n-1; j++)
    if(s[n-1][j] > s[n-1][w[n-1]]) w[n-1] = j;
  for(i=n-2; i>=0; i--){
    if(vPred[i+1][w[i+1]] == 0) w[i] = w[i+1]-1;
    else w[i] = w[i+1];
  }
}

void writeData(int s[][100], int w[], int n){
  ofstream out("triunghi.out");
  out << " Suma maxima = " << s[n-1][w[n-1]] << endl;
  for(int i=0; i<n; i++){
    out << w[i]+1 << endl;
  }
}
int main(){
  int i, j, n;
  int a[100][100], s[100][100], vPred[100][100];
  int w[100];
  readData(a, n);
  doProcess(a, n, s, vPred, w);
  writeData(s, w, n);
  return 0;
}
```

vPred[*i*][0] este mereu 1!

Suma optimală care se termină cu *a*[*i*][0] va fi întotdeauna *a*[*i*][0] + *s*[*i* – 1][0]!

Pentru o pereche (*i*, *j*), *i*, *j* ≠ 0 are loc:

– **dacă** *s*[*i* – 1][*j* – 1] ≥ *s*[*i* – 1][*j*], atunci suma maximală cu ultim element *a*[*i*][*j*] a fost obținută printr-un pas spre dreapta (cu ajutorul *a*[*i* – 1][*j* – 1])

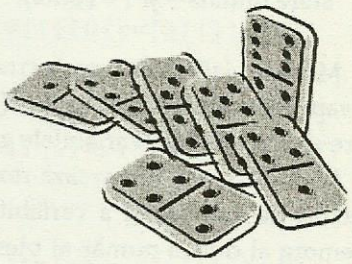
– **altfel**, suma maximală cu ultimul element *a*[*i*][*j*] a fost obținută cu ajutorul unui pas direct-jos (cu elementul *a*[*i* – 1][*j*]).

Pentru ambele cazuri *vPred*[*i*][*j*] și *s*[*i*][*j*] vor fi calculate corespunzător.

Probleme propuse

1. Câte drumuri corecte sunt pentru un anumit n dat? Câte dintre acestea conțin o anumită poziție dată (i, j)?
2. Se poate să existe mai multe drumuri care conduc la obținerea sumei optime. Dezvoltați programul astfel încât să fie afișate toate aceste drumuri.
3. Presupunem că atunci când se trece la linia următoare este permis şi un pas spre stânga-jos. Modificați programul pentru problema astfel transformată.
4. Pentru reprezentarea datelor am folosit tablourile bidimensionale *a*[][], *s*[][] şi *vPred*[][], dar de fapt se folosesc doar zonele de sub diagonala principală. Pentru fiecare astfel de tablou, se alocă 100 × 100 elemente, când de fapt sunt necesare mult mai puține. Folosiți pentru stocarea dinamică a celor $\frac{n(n+1)}{2} = 1 + 2 + ... + n$ elemente un *std::vector*. Modificați programul astfel încât să utilizați doar *std::vector* în locul tuturor tablourilor.

Problema 5. Domino



Se consideră un şir de piese de domino. Fiecare piesă poate fi rotită în jurul centrului ei cu 180°. Să se determine subşirul de piese de lungime maximă în care oricare două piese alăturate au înscrise acelaşi număr: al doilea număr de pe prima piesă coincide cu primul număr înscris pe cea de-a doua.

Date de intrare. În fişierul *domino.in* se găsesc cel mult 200 de piese de domino, câte o pereche pe linie. Fiecare pereche conține numere între 0 şi 6.

Date de ieşire. În fişierul *domino.out* trebuie afişat pe prima linie numărul maxim de piese al subşirului cu proprietatea din enunțul problemei, iar pe următoarele linii, câte una pe linie, trebuie scrise piesele în ordine. Exemplu:

domino.in	domino.out
5 6 1 1 2 5 3 2 1 5 6 4 5 1 5 3 2 1 1 4 5 5 5 2 3 3	6 6 5 5 2 2 3 3 5 5 5 5 2

(din [Olt00])

Analiza problemei şi proiectarea soluției

Definim tipul *TPiece*, ce reprezintă o piesă şi construim două tablouri:

- *v*[200][2] cu semnificația:
  - *v*[*i*][0] = lungimea maximă a unui subşir de piese care începe cu piesa *i* nerotită,
  - *v*[*i*][1] = lungimea maximă a unui subşir de piese care începe cu piesa *i* rotită;
- *vNext*[200][2], cu semnificația:
  - *vNext*[*i*][0] = poziția succesorului piesei *i* în subşirul maximal care începe cu piesa *i* nerotită,
  - *vNext*[*i*][1] = poziția succesorului piesei *i* în subşirul maximal care începe cu piesa *i* rotită.

Utilizăm variabilele:

- *iMax* = indexul primei piese al subşirului maximal curent,
- *bit* (0 sau 1) = starea primei piese în subşirul maximal reprezentată de *iMax* (0 – stare inițială – şi 1 – rotită).

Metoda *doProcess()* prelucrează secvențial piesele de domino de la stânga spre dreapta şi calculează valorile *v*[*i*][0] şi *v*[*i*][1] pe baza numerelor deja calculate, după care se actualizează variabilele globale *iMax* şi *bit*.

Metoda *write()* furnizează rezultatele şi le scrie în fişierul de ieşire pe baza celor doi vectori de mai sus şi a variabilelor globale *iMax* şi *bit*. În variabila locală *aux* se va memora al doilea număr al piesei curente în subşirul maximal pentru a şti ce poziție are următoarea piesă (cunoaştem această piesă cu ajutorul vectorului *vNext*[][]).

Program

```
#include <fstream>
#include <vector>
#include <utility>

using namespace std;

typedef pair<short, short> TPiece;

vector<TPiece> vP;
int iMax, bit;
int vNext[200][2], v[200][2];

void read(){
  ifstream in("domino.in");
  TPiece p;
  while(in && !in.eof()){
    in >> p.first >> p.second;
```

```
    vP.push_back( p );
  }
}

void doProcess(){
  int i, j, auxMax;
  int n = (int)vP.size();
  v[n-1][0] = 1; vNext[n-1][0] = n;
  v[n-1][1] = 1; vNext[n-1][1] = n;
  iMax = n-1; bit = 0;
  for(i=n-2; i>=0; i--){
    v[i][0] = 1; vNext[i][0] = n;
    v[i][1] = 1; vNext[i][1] = n;
    for(j=i+1; j<n; j++){
      if(vP[j].first==vP[i].second && v[i][0]<v[j][0]+1){
        v[i][0] = v[j][0]+1;
        vNext[i][0] = j;
      }
      if(vP[j].second==vP[i].second && v[i][0]<v[j][1]+1){
        v[i][0] = v[j][1]+1;
        vNext[i][0] = j;
      }
      if(vP[j].first==vP[i].first && v[i][1]<v[j][0]+1){
        v[i][1] = v[j][0]+1;
       vNext[i][1] = j;
      }
      if(vP[j].second==vP[i].first && v[i][1]<v[j][1]+1){
        v[i][1] = v[j][1]+1;
        vNext[i][1] = j;
      }
    }
    auxMax =  v[i][0]>v[i][1]?v[i][0]:v[i][1];
    if(auxMax > v[iMax][bit]){
      iMax = i;
      bit = v[i][0]>=v[i][1]?0:1;
    }
  }
}

void write(){
  int aux;
  ofstream out("domino.out");
  out << v[iMax][bit] << endl;
  int n = (int) vP.size();
  if(0==bit){
    aux = vP[iMax].second;
    out << vP[iMax].first << " " << vP[iMax].second << endl;
```

Folosim instrucțiunea de decizie „*?:*", care este o alternativă pentru construcții de forma „*if(A) then B else C;*". Dacă *A* este diferit de zero sau de *NULL*, atunci rezultatul este *B*, altfel este *C*.

```
  }
  else{
    aux = vP[iMax].first;
    out << vP[iMax].second << " " << vP[iMax].first << endl;
  }
  iMax = vNext[iMax][bit];
  while(iMax != n){
    if(aux == vP[iMax].first){
      out << vP[iMax].first << " " << vP[iMax].second << endl;
      aux = vP[iMax].second;
      bit=0;
    }
    else{
      out << vP[iMax].second << " " << vP[iMax].first << endl;
      aux = vP[iMax].first;
      bit=1;
    }
    iMax = vNext[iMax][bit];
  }
}

int main(){
  read();
  doProcess();
  write();
  return 0;
}
```

Probleme propuse

1. Modificați programul astfel încât să nu mai folosească variabile globale.
2. Găsiți o alternativă mai scurtă pentru cele patru instrucțiuni de decizie *if*.
3. Rezolvați problema pentru cazul când se cere subsecvența maximală (piesele trebuie să fie consecutive în șirul inițial).

Problema 6. Împărțirea cadourilor

Doi frați au primit de Crăciun multe cadouri, care însă nu aveau pe ele etichete cu numele lor, ci doar prețurile. Așa că, atunci când frații s-au trezit, au intrat într-o mare dilemă fiindcă nu știau cum să își împartă cadourile Moșului. Știind că fiecare cadou are o valoare cuprinsă între 1 și 100 inclusiv și că sunt maxim 50 de cadouri pentru ambii frați, scrieți un program care să împartă cât mai echitabil

cadourile, astfel încât sumele să fie cât mai apropiate.

Date de intrare. În fişierul *cadouri.in* se găsesc informaţiile referitoare la cadouri: pe prima linie numărul total de cadouri, pe următoarea linie valorile lor.

Date de ieşire. În fişierul *cadouri.out* trebuie scrise două sume care sunt cele mai apropiate, corespunzătoare unei împărţiri a cadourilor, pe a doua linie valorile corespunzătoare cadourilor care însumează primul număr găsit, pe a treia linie, valorile corespunzătoare cadourilor care însumează al doilea număr găsit. Exemple:

cadouri.in	cadouri.out
7 28 7 11 8 9 7 27	48 49 28 11 9 7 8 7 27
15 12 43 8 90 13 5 78 34 1 97 31 65 80 15 17	294 295 12 43 90 13 5 34 97 8 78 1 31 65 80 15 17

(CEOI, Ungaria, 1995)

Analiza problemei şi proiectarea soluţiei

O primă idee de rezolvare ar fi generarea tuturor submulţimilor de cadouri şi selectarea celei mai convenabile. Această variantă este însă ineficientă din cauză că are o complexitate exponenţială.

Cele n preţuri ale cadourilor vor fi salvate în tabloul *aPresent*[]. O altă variantă de rezolvare ar fi construirea unui vector *aS*[] cu semnificaţia:

- $aS[0] = 0$;
- $aS[j] = i$, pentru $i \in \{1, ..., n\}$, aceasta însemnând că ultima valoare introdusă pentru a obţine un total j este $aPresent[i]$;
- $aS[j] = 51$, atunci când suma j nu poate fi obţinută.

Variabila *sum* conţine suma tuturor cadourilor. Atunci când cadoul $aPresent[i]$ va fi adăugat la subtotalul lui j se obţine o nouă valoare $j + aPresent[i]$. Au loc: $aS[j] < i$ (un cadou poate fi adăugat unei sume cel mult o singură dată şi vom adăuga cadourile în ordine crescătoare). Prin secvenţa:

```
for(i=0; i<=5000; i++) aS[i] = 51;
aS[0] = 0;
for(i=1; i<=n; i++)
  for(j=0; j<=sum/2-aPresents[i]; j++)
    if(aS[j]<i &&  aS[j+aPresents[i]]==51)
       aS[j+aPresents[i]] = i;
```

se vor calcula valorile vectorului *aS*[]. În final, ne interesează să găsim cea mai apropiată sumă de mijloc; aceasta va fi una dintre sume şi putem face afişarea celor

două sume optime. Pentru a reconstitui valorile celor două mulțimi de cadouri, vom scrie o funcție recursivă *recoverPresents()*. Ea va marca cu 1 în tabloul *a*[] cadourile considerate în prima mulțime, cu ajutorul căruia se construiește și a doua mulțime (cadourile rămase sunt marcate în *a*[] cu 0):

```
if(aS[i]>0){
    recoverPresents(i-aPresents[aS[i]], out);
    out << aPresents[aS[i]] << " ";
    a[aS[i]]=1;
}
```

Program

```
#include <fstream>

using namespace std;

short aS[5001];
short aPresents[51], a[51];
short n, i;
int j, sum=0;

void read(){
  ifstream in("cadouri.in");
  in >> n;
  for(i=1; i<=n; i++){
    in >> aPresents[i];
    sum += aPresents[i];
    a[i]=0;
  }
}

void recoverPresents(int i, ofstream& out){
  if(aS[i]>0){
    recoverPresents(i - aPresents[aS[i]], out);
    out << aPresents[aS[i]] << " ";
    a[aS[i]]=1;
  }
}

void write(){
  ofstream out("cadouri.out");
  for(j=sum/2; j>=0; j--)
    if(aS[j] != 51){
      out << j << " " << sum - j << endl;
      recoverPresents(j, out);
      out << endl;
```

```
      break;
    }
    for(i=1; i<=n; i++)
      if(!a[i])
        out << aPresents[i] << "  ";
}

int main(){
  read();
  for(i=0; i<=5000; i++) aS[i]=51;
  aS[0] = 0;
  for(i=1; i<=n; i++)
    for(j=0; j<= sum/2 - aPresents[i]; j++)
      if(aS[j]<i &&  51==aS[j+aPresents[i]])
        aS[j+aPresents[i]] = i;
  write();
  return 0;
}
```

Probleme propuse

1. Modificați programul astfel încât să nu mai utilizeze variabile globale.
2. Modificați programul astfel încât în locul tuturor tablourilor să folosiți *std::vector*.
3. Reprezentați datele ca o listă cu perechi *(sumă, index_ultim_element_introdus)*, care inițial e vidă și crește secvențial; odată cu procesarea unui nou element, se vor reactualiza/adăuga noi perechi la listă.

Problema 7. Sume asemănătoare

Fie un număr natural n și o mulțime M de numere naturale. Găsiți o submulțime U a lui M care îndeplinește condiția că ultimele trei cifre ale sumei elementelor sale coincid cu ultimele trei cifre ale lui n. De exemplu, pentru n = 4569 și M = {45345, 65892, 78678, 111153, 190223, 675451, 543876, 23980, 99453, 990121, 656555, 432908, 12361}, o astfel de submulțime poate fi U = {23980, 675451, 190223, 78678, 65892, 45345}:

$$23980 + 675451 + 190223 + 78678 + 65892 + 45345 = 1079569$$

Pentru n = 8 și M = {65968, 65432, 65439876, 9078655, 56743, 54321, 6543298, 453129, 54321, 9999999, 567543}, U ar putea fi {9999999, 6543298, 56743, 65968}:

$$9999999 + 6543298 + 56743 + 65968 = 16666008$$

Date de intrare. Pe prima linie a fişierului *numere.in* se află numărul natural *n*, urmat de elementele mulțimii *M* (toate în intervalul [0, 2000000000] şi *M* are cel mult 5000 de elemente).

Date de ieşire. Scrieți în fişierul *numere.out* elementele mulțimii *U* care satisfac condiția din enunț. Dacă *n* are mai puțin de trei cifre atunci i se vor adăuga înainte numărul necesar de zerouri. Dacă nu există soluție, atunci scrieți în fişierul de ieşire: *nu există soluție!* Exemplu:

numere.in	numere.out
4569 45345 65892 78678 111153 190223 675451 543876 23980 99453 990121 656555 432908 12361	23980 675451 190223 78678 65892 45345
8 65968 65432 65439876 9078655 56743 6543298 453129 54321 9999999 567543	9999999 6543298 56743 65968
45678 65432 65439876 9078655 56743 54321 6543298 453129 54321 9999999 567543	nu exista solutie!

Analiza problemei şi proiectarea soluției

Problema este foarte asemănătoare cu problema anterioară 5.

Construim vectorul *vLast* cu 1000 de elemente, care sunt inițializate cu –1. Elementele vectorului sunt actualizate secvențial, pe măsura prelucrării elementelor mulțimii date. Semnificația elementelor sale este:

- *vLast*[*xyz*] conține –1, atunci când nu a fost încă găsită nici o sumă ce are ca ultime trei cifre *xyz*;
- *vLast*[*xyz*] conține indexul *i* al elementului ultim adăugat (cu cel mai mare index) pentru a obține o sumă ale cărei ultime trei cifre sunt *xyz*.

Dacă, de exemplu, primul număr al mulțimii *M* este 45345, atunci *vLast*[345] se va seta pe 0 (indexul primului element al mulțimii).

Citim elementele mulțimii *M* în vectorul *vElem*. Elementele citite vor fi procesate iterativ (*aux* ← *vElem*[*i*], pentru *i* = 0, 1, 2, ...):

- dacă nu a fost găsită nici o sumă ale cărei ultime trei cifre coincid cu ultimele trei cifre ale lui *aux* (*vLast*[*aux*%1000] == –1), atunci salvăm indexul elementului *i* actual în *vLast*[*aux*%1000]:

```
aux = vElem[i];
if( vLast[aux%1000]==-1 ){
    vLast[aux%1000] = i;
}
```

- pentru toate sumele *j* deja calculate (*vLast*[*j*] ≠ –1) calculăm posibilele seturi de trei cifre terminale ale sumelor pe care le putem obține prin adăugarea lui *aux* (*sum* ← (*aux* + *j*)%1000). Dacă valoarea corespunzătoare în *vLast* este încă –1, atunci ea se va acualiza cu *i*:

```
for(j=0; j<1000; j++){
  if(vLast[j]!=-1 && j!= aux%1000 && vLast[j]!=i){
    int sum = (aux+j)%1000;
    if(vLast[sum]==-1){
      vLast[sum] = i;
    }
  }
}
```

Dacă un element din vectorul *vLast* a fost suprascris după inițializarea cu –1, atunci el nu se va mai schimba. Cu ajutorul vectorului *vLast* vom regăsi termenii sumei cerute:

```
while( vLast[i]>=0 ){
  out << " " << vElem[vLast[i]];
  j=i;
  if(i < vElem[vLast[i]]%1000){
    i = 1000+i-vElem[vLast[i]]%1000;
  }else {
    i = i-vElem[vLast[i]]%1000;
  }
    vLast[j]= -1;
}
```

Dacă *sum* = ...127 şi ultimul termen al sumei ...543 (127 > 543), atunci *sum* ← ...584 (1127 · 543).

Dacă *sum* = ...543 şi ultimul termen al sumei = ...127 (127 < 543), atunci *sum* ← ...416 (543 – 127).

Program

```
#include <fstream>
#include <vector>

using namespace std;

int main(){
  vector<unsigned long long>  vElem, v;
  vector<int> vLast;
  long aux;
  int start, i, j;
  ifstream in("numere.in");
  ofstream out("numere.out");
  in >> start;
```

```
  while(in && !in.eof()){
    in >> aux;
    vElem.push_back(aux);
  }

  for(i=0; i<1000; i++)
    vLast.push_back(-1);

  for(i=0; i<vElem.size(); i++){
    aux = vElem[i];
    if(-1==vLast[aux%1000]){
      vLast[aux%1000] = i;
    }
    for(j=0; j<1000; j++){
        if(vLast[j]!=-1 && j!= aux%1000 && vLast[j]!=i){
          int sum = (aux + j)%1000;
          if(-1==vLast[sum]){
            vLast[sum] = i;
          }
        }
    }
  }

  i=start;
  if(i!=0) vLast[0] = -1;
  if(vLast[i] != -1){

    while(vLast[i]>=0){
      out << " " << vElem[vLast[i]];
      j=i;
      if(i<vElem[vLast[i]]%1000){
         i = 1000+i-vElem[vLast[i]]%1000;
      }else {
        i = i-vElem[vLast[i]]%1000;
      }
      vLast[j]= -1;
    }

  }
  else out << "nu exista solutie!";
  return 0;
}
```

Probleme propuse

1. Câte posibilități de a obține o sumă cu trei cifre ce coincid cu cele ale lui n există? Dezvoltați programul astfel încât să scrie în fișierul de ieșire mai întâi numărul de sume posibile, urmat de una dintre ele:

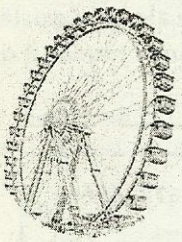
numere1.in	numere1.out
4569 45345 65892 78678 111153 190223 675451 543876 23980 99453 990121 656555 432908 12361 65432 453219	...569: 7 23980 675451 190223 78678 65892 45345
781 65968 65432 65439876 9078655 56743 6543298 453129 54321 9999999 567543 4565 67543 45365 34546	...781: 5 567543 54321 6543298 56743 65439876

   Subșirul indecșilor elementelor în submulțimea determinată este cel mai mic lexicografic relativ la toate posibilitățile de a obține o sumă corectă. Schimbați programul astfel încât să se scrie în fișierul de ieșire suma cu subșirul indecșilor cea mai mare lexicografic.
2. Modificați programul astfel încât să se determine o sumă ce are număr maximal de termeni. Modificați apoi astfel încât să se scrie una cu număr minimal de termeni.
3. Adăugăm și posibilitatea de a scădea termeni. Modificați programul corespunzător acestei extensii.
4. Modificați programul astfel încât numerele date (inclusiv n) să poată fi și negative.
5. Fie date n, k și elementele mulțimii M (toate din $\mathbb{N}$; $0 \leq k \leq 1000$, celelalte mai mari sau egale cu 0 și mai mici sau egale cu 2.000.000.000). Găsiți o submulțime U a mulțimii date M cu proprietatea că ultimele trei cifre ale sumei puterilor de ordin k a elementelor lui U să coincidă cu cele ale lui n.
6. Formulăm o nouă problemă în care elementele mulțimii M sunt numere reale pozitive cu maxim trei zecimale după virgulă. Ne interesează o submulțime a lui m cu proprietatea că suma elementelor sale să fie un număr natural. Scrieți un nou program ce rezolvă problema. Exemplu:

numere2.in	numere2.out
0.123 6.789 **10.125** 115.117 11.03 12.12 **7511.1 315.79** 113.02 **17.985**	10.125 7511.1 315.79 17.985

7. Este permisă și scăderea, iar numerele au voie să fie și negative. Modificați programul pentru aceste condiții în suplimentare.

Problema 8. Scoțieni la Oktoberfest



Într-o seară, un grup de scoțieni sunt în vizită la Oktoberfest și bineînțeles că vor să bea bere. Ei și-au procurat de ceva mai multă vreme bilete de intrare, care însă, din păcate, erau pentru două corturi diferite (au fost cumpărate prea târziu). Numărul de scoțieni este $2n$ și au pentru fiecare dintre cele două corturi câte n bilete, deci trebuie să se împartă în două grupe. Pentru a evita cearta și discuțiile nesfârșite, au hotărât să dea cu banul. Ei au decis să arunce pe rând: cine aruncă pajură va merge în al doilea cort, altfel, va merge în primul cort. Normal că ei încetează cu aruncatul atunci când toate biletele pentru unul dintre corturi s-au epuizat. Ultimii doi la aruncat moneda sunt cei doi buni prieteni, Ian și Alistair, și ei își doresc foarte tare să meargă împreună în același cort, indiferent care. Ce probabilitate W există ca dorința prietenilor să se împlinească?

Întrebare. Pentru a calcula probabilitatea cerută, analizăm starea intermediară după $2n-2$ aruncări. Rămân în acest moment două bilete identice pentru unul din cele două corturi?
Date de intrare. În fișierul *oktoberfest.in* se găsesc mai multe numere naturale, fiecare cuprins între 1 și 500 și reprezentând pe n (este vorba despre $2n$ scoțieni).
Date de ieșire. Scrieți în fișierul *oktoberfest.out* pentru fiecare caz de intrare soluția în forma: n: W, unde n reprezintă o jumătate din numărul scoțienilor și probabilitatea W este scrisă cu trei zecimale exacte. Exemplu:

oktoberfest.in	oktoberfest.out
1	1: 0.0000
2	2: 0.5000
3	3: 0.6250
4	4: 0.6875
5	5: 0.7266
6	6: 0.7539
56	56: 0.9241
345	345: 0.9696
432	432: 0.9728
500	500: 0.9747

(după ACM North-Western European Regionals, 1996, Problem A. Burger)

Analiza problemei și proiectarea soluției

Probabilitatea ca dintr-o aruncare a monedei să se obțină cap este $\frac{1}{2}$. Este mai simplă însă determinarea probabilității p ca Ian și Alistair să nu ajungă în același cort. Probabilitatea ca ei să meargă în același cort va fi $1-p$. Notăm cu $P(m, s)$

probabilitatea ca din m aruncări ale monedei să apară de s ori cap. Pentru cazurile $s = 0$ şi $s = m$, probabilitatea $P(m, s) =$ este $\frac{1}{2^m}$:

$$P(m, 0) = P(m, m) = \frac{1}{2^m}$$

Explicația este că probabilitatea ca de la prima aruncare să apară cap este $\frac{1}{2}$, probabilitatea să apară cap şi din a doua aruncare este $\frac{1}{2} \cdot \frac{1}{2}$, iar de a apărea şi în a treia este $\frac{1}{4} \cdot \frac{1}{2} = \frac{1}{2^3}$. Această probabilitate este egală cu probabilitatea ca dintre toți vectorii de lungime m cu elemente 0 şi 1 să îl alegem pe acela care conține doar zerouri, respectiv doar elemente 1 (există numai un astfel de vector şi numărul total al vectorilor este 2^m). Dacă $s \neq 0$ şi $s \neq m$, atunci are loc formula recursivă:

$$P(m, s) = \frac{1}{2} P(m-1, s) + \frac{1}{2} P(m-1, s-1)$$

Posibilitatea ca din m aruncări să obținem de s ori cap este	$P(m, s) =$
să se arunce pajură după ce din $m-1$ aruncări a fost de s ori cap	$\frac{1}{2} P(m-1, s)$
sau	+
să se arunce cap după ce din $m-1$ aruncări a fost de $(s-1)$ ori cap	$\frac{1}{2} P(m-1, s-1)$

Aşadar:

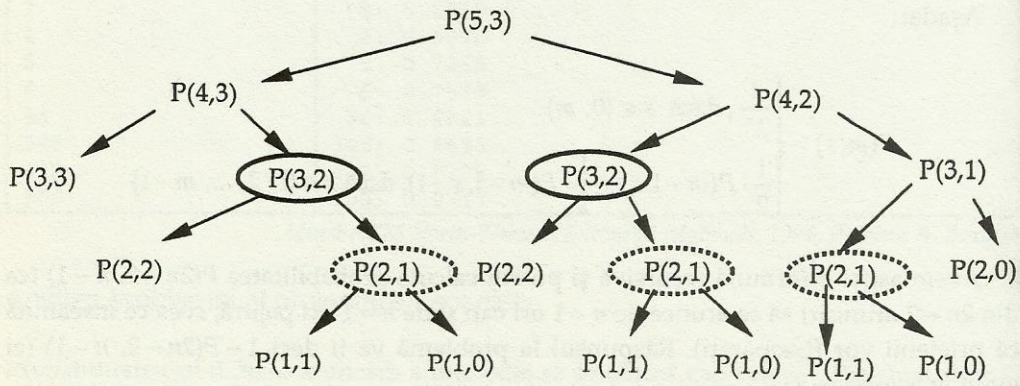
$$P(m,s) = \begin{cases} \frac{1}{2^m}, \text{dacă } s \in \{0, m\} \\ \frac{1}{2} \cdot P(m-1,s) + \frac{1}{2} \cdot P(m-1,s-1), \text{dacă } s \in \{1, 2, \ldots, m-1\} \end{cases}$$

Avem astfel o formulă recursivă şi putem calcula probabilitatea $P(2n-2, n-1)$ (ca din $2n-2$ aruncări să se arunce de $n-1$ ori cap şi de $n-1$ ori pajură, ceea ce înseamnă că prietenii vor fi separați). Răspunsul la problemă va fi deci $1 - P(2n-2, n-1)$ (ei merg în acelaşi cort).

Program 1. O primă variantă recursivă:

```
#include <iostream>

double f(int n){
  double r=1;
  for(int i=0; i<n; i++) r*=0.5;
  return r;
}

double P(int m, int q){
  if(q==0 || q==m) return f(m);
  return  0.5*P(m-1, q) + 0.5*P(m-1, q-1);
}

int main(){
  int n;
  std::cout << "Numar perechi: ";
  std::cin  >> n;
  std::cout << "Probabilitatea> ";
  std::cout.precision( 4 ); std::cout << 1 - P(2*n-2,n-1);
  return 0;
}
```

Această variantă are însă un mare neajuns: timpul de execuție! Vom vedea că timpul de execuție pentru $n > 12$ crește considerabil. Aceasta se întâmplă din cauza faptului că aceleaşi valori (ca şi în cazul şirului lui Fibonacci) se calculează independent de mai multe ori (algoritmul este exponențial). Arborele recursiv pentru $P(5, 3)$:



Acest algoritm arată încă o dată că folosirea unei metode recursive nu este întotdeauna cea mai bună alegere.

Deoarece numărul n al perechilor poate creşte până la 500, este evident că trebuie să găsim o altă metodă de rezolvare. Putem să construim un tablou bidimensional $p[0..m][0..m]$, cu semnificația că $p[i][j]$ este probabilitatea ca din i aruncări să se obțină de j ori cap.

```
p[0][0]
p[1][0]  p[1][1]
p[2][0]  p[2][1]  p[2][2]
p[3][0]  p[3][1]  p[3][2]  p[3][3]
.....................................

p[m][0]  p[m][1]  p[m][2]  ......p[m][m]
```

Program 2

```
#include <fstream>

const int NMAX = 500;
using namespace std;

int main(){
  int n, i, j;
  float p[2*NMAX+1][2*NMAX+1];
  ifstream fin("oktoberfest.in");
  ofstream fout("oktoberfest.out");
  p[0][0] = 1;
  p[1][0] = p[1][1] = 0.5;
  for(i=2; i<=2*NMAX-2; i++){
    p[i][0] = p[i][i] = p[i-1][0]*0.5;
    for(j=1; j<i; j++)
      p[i][j]=0.5*p[i-1][j]+0.5*p[i-1][j-1];
  }
  while(fin && !fin.eof()){
    fin  >> n;
    fout.width(3);
    fout <<  n << ":";
    fout.precision(4);
    fout.flags(ios::fixed);
    fout << 1-p[2*n-2][n-1] << endl;
  }
  return 0;
}
```

La început, se calculează valorile de sub diagonala principală a tabloului $p[][]$, linie cu linie, astfel încât pentru determinarea valorilor dintr-o linie se folosesc cele din linia anterioară. Această metodă nu mai are probleme cu timpul de execuție, însă are o altă

deficiență majoră: spațiul folosit. Pentru *NMAX* = 500, tabloul *p*[][] are nevoie de 1002101 de tip *float*. De obicei, un element de tip *float* are 4 octeți, ceea ce înseamnă că avem nevoie de 4 · 1002001 = 4008004 octeți (aproximativ 3,9 MB). La execuția programului, putem obține o eroare de genul: *„Unhandled exception at 0x00435415 in oktoberfest.exe: 0xC00000FD: Stack overflow"*.

Pentru a evita și acest lucru, putem folosi doi vectori (unul pentru valorile din linia predecesoare și unul pentru cea actuală), care vor fi mereu actualizați, și un vector *c*, care conține valorile *P(i, i/2)* pentru *i* număr par. Mai mult, putem folosi doar doi vectori, unul cu probabilitățile *P(i, i/2)* și un altul ajutător, care conține mereu probabilitățile *P(i, 0)*, *P(i, 1)* până la *P(i, i)* pentru linia *i* curentă. Astfel, obținem următoarea variantă:

Program 3

```
#include <fstream>
#include <vector>

const int NMAX = 500;
using namespace std;

int main(){
  int n, i, j;
  float  aux1, aux2;
  vector<float> a, c;
  ifstream fin("oktoberfest.in");
  ofstream fout("oktoberfest.out");
  a.push_back(0.5);
  a.push_back(0.5);
  c.push_back(1);
  for(i=2; i<=1000; i++){
    aux1 = a[0];
    a[0] = (float)0.5*a[0];
    for(j=1; j<i; j++){
      aux2 = a[j];
      a[j] = (float)(0.5*aux1 + 0.5*a[j]);
      aux1=aux2;
    }
    a.push_back(a[0]);
    if(0==i%2) c.push_back(a[i/2]);
  }

  while(fin && !fin.eof()){
    fin >> n;
    fout.width(3);
    fout  << n << " perechi: ";
    fout.precision(4);
```

```
    fout.flags(ios::fixed);
    fout << 1-c[n-1] << endl;
  }
  return 0;
}
```

Altă metodă de rezolvare

Observăm că formula pentru $P(m, q)$ este foarte asemănătoare cu formula recursivă pentru combinări (coeficienții binomiali). Pentru rezolvarea problemei, avem de fapt nevoie doar de valorile $P(2n, n)$, pentru $n \in \{0, 1, 2, \ldots, 500\}$. Notăm $P(2n, n)$ cu T_n și considerăm următoarele mulțimi:

- M_1 = mulțimea tuturor vectorilor de lungime $2n$, care conțin doar 0 și 1 (0 și 1 au semnificația de cap, respectiv pajură);
- M_2 = mulțimea vectorilor de $2n$ care conțin n elemente 0 și n elemente 1, cu condiția ca valorile ultimelor două poziții să fie diferite.

Cu această notație, T_n este $\frac{|M_2|}{|M_1|}$.

$|M_1|$ este 2^{2n}, $|M_2|$ este jumătate din C_{2n}^n, de unde rezultă: $T_n = \frac{C_{2n}^n}{2^{2n+1}}$. (1)

Din (1) rezultă reprezentarea recursivă pentru șirul T_n:

$$T_n = \frac{2n-1}{2n} T_{n-1}, \text{ pentru } n > 0 \text{ și } T_0 = 1 \quad (2)$$

Pe baza formulei (2), scriem un nou program:

Program 4

```
#include <fstream>
#include <vector>

const int NMAX = 500;
using namespace std;

int main(){
  int n, i, j;
  float aux1, aux2;
  vector<float> T;
  ifstream fin("oktoberfest.in");
  ofstream fout("oktoberfest.out");
  T.push_back(1);
```

$$T_n = \frac{C_{2n}^n}{2^{2n+1}}$$

$$T[0] \leftarrow 1$$

$$T[i] \leftarrow \frac{2i-1}{2i} T[i-1]$$

```
  for(i=1; i<=NMAX; i++){
    T.push_back((float)((2*i-1.0)/(2*i)*T[i-1]));
  }

  while(fin && !fin.eof()){
    fin >> n;
    fout.width(3);
    fout  << n << ": ";
    fout.precision(4);
    fout.flags(ios::fixed);
    fout << 1-T[n-1] << endl;
  }
  return 0;
}
```

Probleme propuse

1. Cât de mare este probabilitatea ca prietenii să ajungă amândoi în cortul 1? Cât de mare este probabilitatea ca Ian să meargă în cortul 1 şi Alistair în cortul 2?
2. În programul 1 folosim funcția *f()* pentru calcularea valorilor $1/2^n$. De ce următoarea variantă nu este corectă?

```
double f(int n){
  long int p = 1;
  for(int i=0; i<n; i++)p*=2;
  return 1/p;
}
```

3. În al doilea tablou, declarăm un tablou bidimensional *p*[][], dar este folosit doar domeniul de sub diagonala principală. Optimizați folosirea spațiului cu ajutorul unei reprezentări ca vector numai a acestei zone.
4. Găsiți cel puțin încă o modalitate de a obține formula (2).
5. Explicați de ce $|M_2|$ este o jumătate din C_{2n}^n.
6. Găsiți o relație dintre T_n şi numerele lui Catalan.

Problema 9. Calul pe tabla de şah

O problemă cunoscută este determinarea posibilităților ca un cal să parcurgă o tablă de şah trecând prin fiecare câmp exact o singură dată (problemă rezolvată în capitolele *Greedy* şi *Backtracking*). Vom considera în acest capitol o problemă înrudită: se dau cele două poziții de start, respectiv de final şi dimensiunea *n* a tablei de şah şi se cere cel mai scurt drum al unui cal între cele

două câmpuri date.

Date de intrare. În fişierul *cal.in* se găsesc cinci numere naturale n, l_1, c_1, l_2, c_2, cu semnificaţiile: n este numărul de linii şi coloane al tablei de şah ($4 \le n \le 20$) şi l_1, c_1, l_2, c_2 sunt „coordonatele" câmpurilor de start şi de sfârşit.

Date de ieşire. Scrieţi în fişierul de ieşire *cal.out* mai întâi lungimea celui mai scurt drum şi apoi un astfel de drum de lungime minimă (dacă există mai multe, atunci se va scrie doar unul singur). Exemplu:

cal.in	cal.out
8 1 1 8 7	Cel mai scurt drum: 5 (1, 1)(3, 2)(5, 3)(7, 4)(6, 6)(8, 7)
5 5 2 1 4	Cel mai scurt drum: 2 (5, 2)(3, 3)(1, 4)
15 2 13 14 15	Cel mai scurt drum: 6 (2, 13)(4, 14)(6, 15)(8, 14)(10, 15)(12, 14)(14, 15)

Analiza problemei şi proiectarea soluţiei

Definim *distanţa* (relativ la două poziţii date) ca fiind numărul minim de mutări ale calului pentru a ajunge de la una la alta. Vom construi o tabelă ce reprezintă tabla de şah şi fiecare celulă conţine distanţa de la poziţia de start la câmpul respectiv. Pentru al doilea exemplu din fişierul de intrare, această tabelă va fi:

3	2	3	2	3
2	3	2	3	2
1	2	1	4	3
2	3	2	1	2
3	0	3	2	3

Câmpul încercuit este câmpul de start, iar câmpul final este cel marcat în pătrat. Pentru a mări claritatea algoritmului în program vom folosi variabilele globale:

```
int Table[NMAX][NMAX]; // Tabela distanţelor
int n;                 // Numărul de linii şi coloane ale tablei
int Pred[NMAX*NMAX];   // Vectorul cu predecesori, ce reţine drumul
```

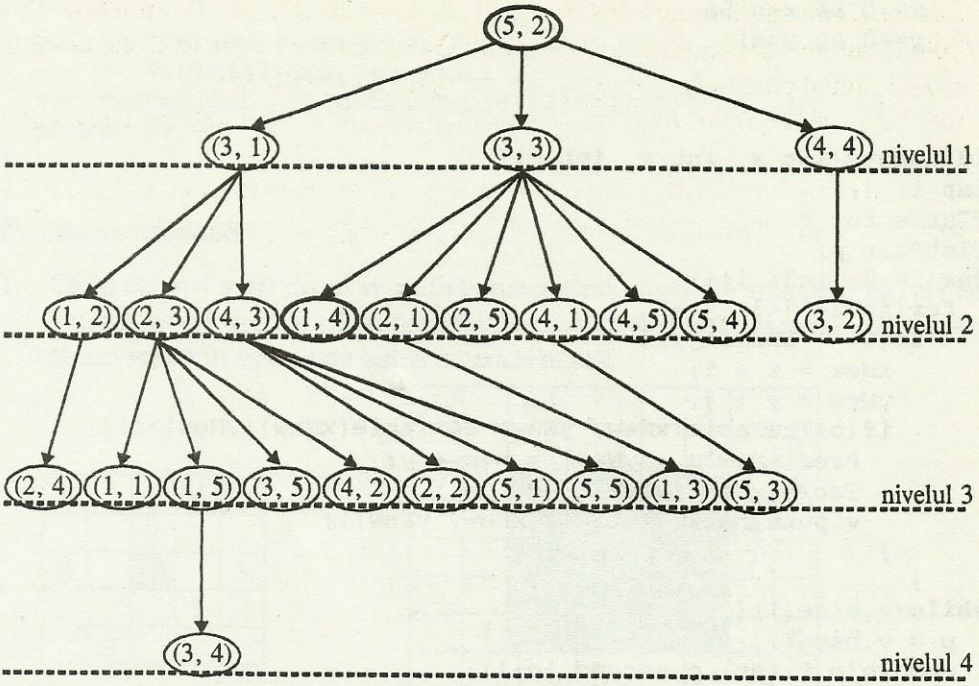
Vom scrie o metodă recursivă *doMove()* care parcurge pătratele tablei începând cu cel iniţial în ordinea dată de pasul calului. În implementarea acestei funcţii, vom folosi:

- *void initTable()*: inițializează fiecare celulă a tabelei cu INT_MAX (un fel de plus infinit); atunci când calul ajunge într-un anumit câmp, vom salva distanța de la câmpul inițial până aici, care sigur va fi mai mică decât INT_MAX;
- *bool onTheTable()*: returnează *true* atunci când punctul (i, j) se află pe tablă, altfel returnează *false;*
- *void recoverWay()*: metodă recursivă care construieşte drumul minim. Cele $n \cdot n$ câmpuri ale tablei sunt salvate într-un vector; de exemplu, pentru $n = 5$:

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

  - câmpul (i, j) va fi salvat în vector la poziția $i \cdot n + j$;
  - pentru un element k din vector, câmpul în tabelă este $(k \text{ div } n, k \text{ mod } n)$;
  - liniile şi coloanele vor fi numerotate intern de la 0; de aceea, scriem în fişierul de ieşire valorile $k \text{ div } n + 1$ şi $k \text{ mod } n + 1$ (în descrierea problemei se începe de la 1!);
- *void doMove()*: metodă recursivă ce are ca parametru un câmp al tablei pentru care distanța a fost deja calculată. Corespunzător acestei poziții, se calculează toate câmpurile încă neatinse şi care pot fi direct succesoare ale câmpului dat ca parametru şi apelează recursiv metoda pentru aceste câmpuri (procesul este finit, deoarece distanțele sunt completate crescător şi calul poate păşi un câmp doar o singură dată).

O altă modalitate de rezolvare ar fi folosirea unui arbore. Nodurile sale reprezintă câmpurile tablei de şah, iar rădăcina este câmpul inițial. Vom introduce câte o muchie între vârfuri (i, j) şi (p, q) dacă se poate păşi cu o mutare a calului din câmpul (i, j) în câmpul (p, q). Nivelurile arborelui pot fi parcurse folosind algoritmul de parcurgere în lățime (*BFS – Breadth First Search*), până când se ajunge la nodul final. Pentru exemplul nostru:



Program

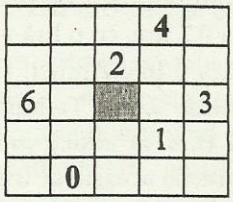
```
#include <fstream>
#include <vector>
#include <cmath>

using namespace std;
const int NMAX = 20;
typedef pair<int, int> TIntPair;
typedef vector<TIntPair> TIntVector;

int Table[NMAX][NMAX];
int n;
int xNew, yNew;
int Pred[NMAX*NMAX];

void initTable(){
  for(int i=0; i<n; i++)
    for(int j=0; j<n; j++)
      Table[i][j]=INT_MAX;
}

bool onTheTable(int x, int y){
  return (
```

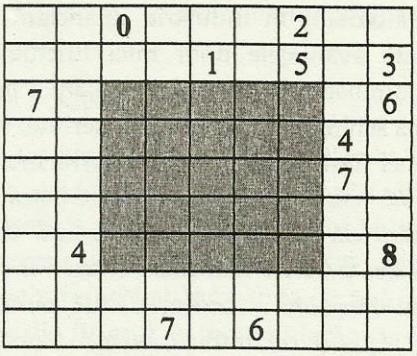
```
      x>=0 && x<n &&
      y>=0 && y<n);

}

void doMove(int x, int y, int c){
  int i, j;
  TIntVector v;
  TIntPair p;
  for(i=-2; i<3; i++)
    for(j=-2; j<3; j++)
      if(2 == abs(i*j)){
        xNew = x + i;
        yNew = y + j;
        if(onTheTable(xNew, yNew) && Table[xNew][yNew]>c){
          Pred[xNew*n + yNew] = x*n + y;
          Table[xNew][yNew] = c;
          v.push_back(TIntPair(xNew, yNew));
        }
      }
  while(v.size()){
    p = v.back();
    doMove(p.first, p.second, c+1);
    v.pop_back();
  }
}

void recoverWay(ofstream &out, int k, int c){
  int i;
  if(c){
    i = Pred[k];
    recoverWay(out, i, c-1);
    out << "(" << k/n + 1 << ", "
        << k%n + 1 << ")";
  }
}

int main(){
  int l1, l2, c1, c2;
  ifstream in("cal.in");
  ofstream out("cal.out");
  in >> n >> l1 >> c1 >> l2 >> c2;
  initTable();
  l1--; l2--; c1--; c2--;
  doMove(l1, c1, 1);
  if( Table[l2][c2] == INT_MAX ){
    out << "nu exista solutie";
    return 0;
```

```
    }
    out << "Cel mai scurt drum: "
        << Table[l2][c2] << endl;
    recoverWay(out, l2*n+c2, Table[l2][c2]+1);
    return 0;
}
```

Probleme propuse

1. Găsiți cel mai scurt drum al calului atunci când acesta are voie doar pe cele două rânduri exterioare ale tablei (câmpurile gri sunt închise). Dacă nu există soluție, atunci scrieți în fișierul de ieșire: *nu exista solutie.*





Două exemple (n = *5 și* n = *9)*

cal1.in	cal1.out
5 5 2 1 4	Cel mai scurt drum: 4 (5, 2)(4, 4)(2, 3)(3, 5)(1, 4)
9 1 3 7 9	Cel mai scurt drum: 8 (1, 3)(2, 5)(1, 7)(2, 9)(4, 8)(2, 7)(3, 9) (5, 8)(7, 9)
8 1 1 8 7	nu exista solutie
15 14 13 1 2	Cel mai scurt drum: 12 (14, 13)(13, 15)(11, 14)(9, 15)(7, 14)(5, 15) (3, 14)(2, 12)(1, 10)(2, 8)(1, 6)(2, 4)(1, 2)

Când nu are soluție problema? Găsiți un drum al calului care acoperă toată această zonă exterioară a tablei (două rânduri), astfel încât fiecare câmp este păşit doar o singură dată (există numai $8n - 16$ câmpuri permise).

2. Scrieți un program care implementează soluția alternativă cu reprezentarea ca arbore şi parcurgerea în lățime.
3. Pentru a găsi toate pozițiile succesoare ale unui câmp dat, folosim metoda *doMove()*. Se dau $n = 5$ şi câmpurile (1, 4), (2, 5) şi (3, 3). Scrieți pe hârtie noile posibile poziții pentru fiecare dintre câmpurile date cu ajutorul buclelor *for*:

```
for(i=-2; i<3; i++)
  for(j=-2; j<3; j++)
    if(2==abs(i*j)) {
      xNew = x + i;
      yNew = y + j;
```

4. *Numărul cailor.* Câți cai se pot amplasa pe o tablă de șah specială de dimensiune $m \times n$ ($1 \le m \le 20$, $1 \le n \le 20$), astfel încât oricare doi să nu se atace între ei?

Problema 10. Fluctuații la bursă

Utilizarea calculatoarelor în industria financiară se extinde și în calculul bursier, pentru a exploata avantajele unor mici fluctuații de preț a mai multor valute. *Arbitrajul* este schimbarea banilor dintr-o valută în alta în scopul obținerii de profit. De exemplu, dacă schimbăm 1,00$ în lire sterline, vom cumpăra 0,7 lire, cu o liră vom cumpăra 9,5 franci francezi și cu un franc francez vom cumpăra 0,16$. Atunci, vom obține $1 \times 0{,}7 \times 9{,}5 \times 0{,}16 = 1{,}064$ $, ceea ce înseamnă un profit de 6,4%. Trebuie să scrieți un program care determină dacă există vreo secvență care să aducă profit, asemănătoare cu cea descrisă mai sus. Pentru a fi o secvență corectă, aceasta ar trebui să înceapă și să se sfârșească cu aceeași valută, putând fi considerată oricare dintre ele.
Date de intrare. Fișierul de intrare *bursa.in* conține una sau mai multe table de conversie. Trebuie să rezolvați problema arbitrajului pentru fiecare tablă în parte. Fiecare tablă este precedată de un număr întreg n ($2 \le n \le 20$) pe o linie care este dimensiunea tablei, urmat de elementele tablei. Presupunem că elementele de pe diagonala principală lipsesc (presupunem că au toate valoarea 1). Primul rând al tablei reprezintă ratele de conversie ale țării 1 relativ la celelalte $n-1$ țări, adică elementul i reprezintă valoarea cu care poate fi obținută o unitate din valuta țării i. Deci fiecare tabelă este definită de $n+1$ linii: una pentru n – numărul de valute – și n linii reprezentând tabela de conversie.
Date de ieșire. Fișierul de ieșire este *bursa.out*. Pentru fiecare tabelă din fișierul de intrare trebuie să determinați dacă există o secvență astfel încât profitul să fie mai mare decât un procent (0,01). Dacă o astfel de secvență există, trebuie să afișați secvența de valute care se schimbă pentru a obține acest profit. Dacă există mai mult de o secvență care va furniza un profit mai mare decât 1%, atunci va trebui să afișați o secvență de lungime minimă, adică una dintre secvențele care folosesc cel mai mic număr de valute pentru a obține profitul. Dacă nu există o secvență de cel mult lungimea n, atunci se va afișa în fișierul de ieșire linia: *Nu exista secvența de arbitraj*! În fișierul de ieșire cazurile se vor numerota, iar formatul va trebui să fie ca în exemplul de mai jos. Exemplu:

bursa.in	bursa.out
3 1.2 0.89 0.88 5.1 1.1 0.15 4 3.1 0.0023 0.35 0.21 0.00353 8.13 200 180.559 10.339 2.11 0.089 0.06111 2 2.0 0.45	Caz 1: Valuta 2 2 -> 1 -> 2 1.056 Caz 2: Valuta 3 3 -> 1 -> 2 -> 3 2.189 Caz 3: Nu exista secvența de arbitraj!

(ACM Internet Programming Contest, 1990)

Analiza problemei şi proiectarea soluției

Vom rezolva problema adaptând un algoritm de tip Roy-Warshall asupra matricei *tabla*[][], care conține inițial tabela de schimb. Pentru a regăsi drumul parcurs în scopul obținerii unei secvențe de arbitraj, vom utiliza un vector bidimensional *pred*[][] cu semnificația: *pred*[*i*][*j*] este predecesorul lui *i* pe drumul de valoare maximă dintre *i* şi *j* (valoarea maximă care se poate obține schimbând valuta *i* în valuta *j*).

- *bool read*() va citi un set de date din fişierul de intrare; dacă mai există un astfel de set, se va returna 1, altfel se va returna 0. În cadrul acestei funcții, se inițializează şi vectorul *pred*[*i*][*j*] = *j* (în acest moment, predecesorul lui *j* pe drumul de la *i* la *j* este *i*), dacă $i \neq j$ şi *pred*[*i*][*i*] = –1.
- *bool gaseste*() conține un algoritm Roy-Warshall modificat; întrucât prima iterație se referă la numărul nodurilor intermediare dintr-un drum, deducem că prima dată când vom obține o valoare pe diagonala principală ≥ 1,01, putem să ne oprim şi să scriem rezultatele (nu ne interesează decât un răspuns).
- void *reconstituieDrum*(*int i, int j*) va reface recursiv drumul între două valute, pe baza vectorului *pred*[].

Program

```
#include <fstream>

#define DIM_MAX 21

int n, nCaz = 0;
float tabla[DIM_MAX][DIM_MAX];
int pred[DIM_MAX][DIM_MAX];
```

```
bool read(std::ifstream& in){
  if(in && !in.eof() && in>>n){
    for(int i=0; i<n; i++){
      tabla[i][i]=1.0;
      for(int j=0; j<n; j++)
        if(i != j){
          in >> tabla[i][j];
          pred[i][j]=i;
        }
        pred[i][i] = -1;
    }
  } else return false;
  return true;
}

void reconstituieDrum(int i, int j, std::ofstream& out){
  int t = pred[i][j];
  if(i!=t){
    reconstituieDrum(i, t, out);
    out << " -> " << t + 1;
  }
}

bool gaseste(std::ofstream& out){
  int x, y, j;
  for(y=0; y<n; y++)
    for(x=0; x<n; x++)
      for(j=0; j<n; j++)
        if(tabla[x][y]*tabla[y][j]>tabla[x][j]){
          pred [x][j] = pred[y][j];
          tabla[x][j] = tabla[x][y] * tabla[y][j];
          if (x==j && tabla[x][j]>=1.01){
            out << "\n Valuta " << x+1
                << std::endl << " " << x+1;
            reconstituieDrum(x, x, out);
            out << " -> " << x+1 << std::endl;
            out.precision(3); //out.width(9);
            out << tabla[x][x];
            return true;
          }
        }
  return false;
}

int main(){
  std::ifstream in("bursa.in");
  std::ofstream out("bursa.out");
  while(read(in)){
```

```
    out << "\nCaz " << ++nCaz << ": ";
    bool ok = gaseste(out);
    if(!ok)
      out << "\n Nu exista secventa de arbitraj!\n";
  }
  return 0;
}
```

Probleme propuse

1. Modificați programul de mai sus astfel încât să se afișeze o secvență de arbitraj care aduce profit maxim (nu este nevoie neapărat ca lungimea să fie maximă).
2. Modificați programul de mai sus astfel încât să se afișeze toate secvențele de arbitraj care aduc un profit de peste 1%, de lungime minimă.

Problema 11. Sume de produse

Se dau numerele naturale $n < 30$ şi $a_1, a_2, ..., a_n$, fiecare mai mic decât 101. Calculați numerele $b_1, b_2, ..., b_n$, astfel încât:

$$b_i = \sum_{1 \le k_1 < k_2 < ... < k_i \le n} a_{k_1} \cdot a_{k_2} \cdot ... \cdot a_{k_i}$$

De exemplu, pentru $n = 3$:

$b_1 = a_1 + a_2 + a_3$

$b_2 = a_1 \cdot a_2 + a_1 \cdot a_3 + a_2 \cdot a_3$

$b_3 = a_1 \cdot a_2 \cdot a_3$

Valorile $b_1, b_2, ..., b_n$ „încap" în tipul *unsigned long long*.

Date de intrare. În fişierul *prodsum.in* se găsesc numerele $a_1, a_2, ..., a_n$.

Date de ieşire. Scrieți numerele $b_1, b_2, ..., b_n$ în fişierul *prodsum.out* (câte unul pe linie) şi înaintea fiecăruia precizați suma cifrelor sale, ca în exemplu:

prodsum.in	prodsum.out
1 2 3 4	1 10 8 35 5 50 6 24
2 1 7 3 4	8 17 6 105 16 295 14 374 15 168

(ACM, South-Eastern European Regionals, 1999, enunț modificat)

Analiza problemei şi proiectarea soluției

Vom folosi relațiile lui Viète.

Dacă $x_1, x_2, ..., x_n$ sunt rădăcinile ecuației:

$$x^n + c_{n-1}x^{n-1} + c_{n-2}x^{n-2} + ... + c_0 = 0 \quad (1)$$

atunci au loc:

$$\begin{aligned}
&x_1 + x_2 + ... + x_n = \sum_{i=1}^{n} x_i = -c_{n-1} \\
&x_1x_2 + x_1x_3 + ... + x_{n-1}x_n = \sum_{\substack{i,j=1 \\ i<j}}^{n} x_ix_j = c_{n-2} \\
&x_1x_2x_3 + x_1x_2x_4 + ... + x_{n-2}x_{n-1}x_n = \sum_{\substack{i,j,k=1 \\ (i<j<k)}}^{n} x_ix_jx_k = -c_{n-3} \quad (2) \\
&\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
&x_1x_2 \cdot ... \cdot x_n = (-1)^n c_0
\end{aligned}$$

Ecuația (1) poate fi scrisă şi sub forma:

$$(x - x_1)(x - x_2) \cdot ... \cdot (x - x_n) = 0 \quad (3)$$

Dacă presupunem că numerele date $a_1, a_2, ..., a_n$ sunt rădăcinile ecuației (3), atunci rezultă, pe baza relațiilor lui Viète, că numerele cerute sunt coeficienții ecuației (1) multiplicați alternativ cu 1 şi –1.

Coeficienții polinomului (1) se vor calcula pas cu pas şi $P[i] = a_i$ pentru orice $i = 0, 1, ..., n$. La pasul al i-lea se va multiplica polinomul actual cu polinomul $(X - a_i)$. Polinomul de start este $X - a_1$, de aceea inițializăm:

– $P[0] = -a[0]$;

– $P[1] = 1$.

Pentru toți k de la 1 la n–1 considerăm următoarea transformare:

$$(X - a[k])(P[0] + P[1]X + ... + P[k-1]X^{k-1} + P[k]X^k). \quad (4)$$

Noii coeficienți ai lui $P[]$ sunt coeficienții calculați simultan pe baza înmulțirii (4):

– $P[0] \leftarrow -P[0] * a[k]$;

– $P[j] \leftarrow P[j-1] - P[j] * a[k]$, pentru orice j de la 1 la k;
– $P[k+1] \leftarrow P[k]$.

Dacă schimbăm semnele conform formulelor lui Viète, atunci numerele cerute b_1, b_2, ..., b_n sunt $(-1)^n P[n]$, $(-1)^{n-1} P[n-1]$, ..., $P[0]$. Ele se vor copia în tabloul $a[]$:

```
j = 1;
if(n%2) j = -1;
for(k=0; k<=n; k++) {a[k] = j*P[k]; j *= -1;}
```

În final, calculăm conform enunțului și suma cifrelor pentru fiecare astfel de număr b_i, folosind funcția *sumDigits()*.

Program

```
#include <fstream>
#include <vector>

using namespace std;

void read(unsigned long long a[31], short& n){
  ifstream in("prodsum.in");
  n = 0;
  while(in && !in.eof()){
    in >> a[n++];
  }
}

void process(unsigned long long a[31], short n){
  short j, k;
  unsigned long long  p[31], q[31];
  if(!(n > 0)) return;
  p[0] = -a[0];
  p[1] = 1;
  for(k=1; k<n; k++){
    q[0] = (-1) * p[0] * a[k];
    for(j=1;j<=k; j++){
      q[j] = p[j-1] - p[j] * a[k];
    }
    q[k+1] = p[k];
    for(j=0; j<=k+1; j++) p[j] = q[j];
  }
  j = 1;
  if(n%2) j = -1;
  for(k=0; k<=n; k++) {a[k] = j*p[k]; j *= -1;}
```

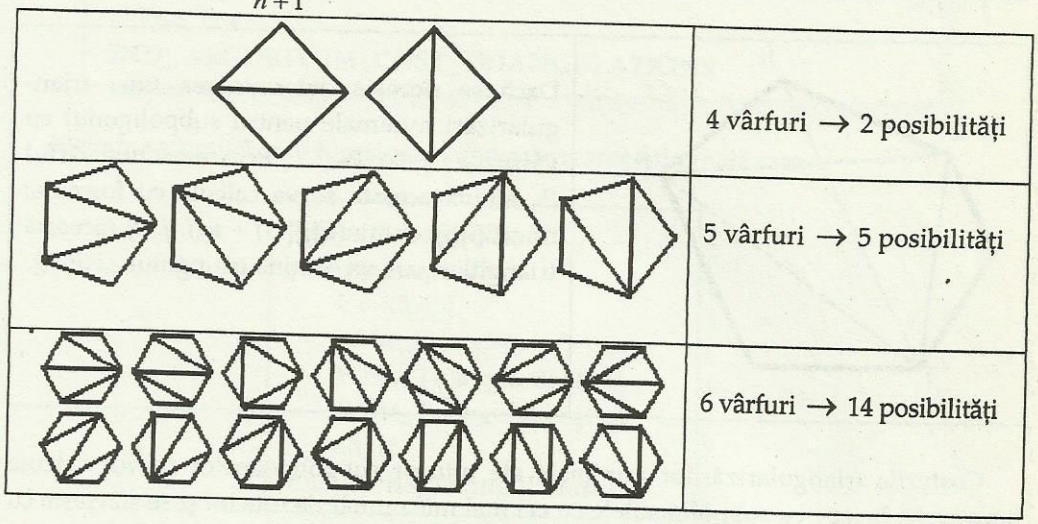
```
}

int sumDigits(unsigned long long n){
  int s = 0;
  while(n){
    s += n%10;
    n /= 10;
  }
  return s;
}

void write(unsigned long long a[31], short n){
  ofstream out("prodsum.out");
  short i;
  for(i=n-1; i>=0; i--){
    out << sumDigits(a[i]) << " ";
    out << a[i] << endl;
  }
}

int main(){
  unsigned long long a[31];
  short n;
  read(a, n);
  process(a, n);
  write(a, n);
  return 0;
}
```

Probleme propuse

1. În program am folosit tablouri pentru a prezenta mai transparent algoritmul. Modificați-l astfel încât să folosiți în locul lor *std::vector*.
2. Condiția ca numerele $b_1, b_2, ..., b_n$ să „încapă" în tipul *unsigned long long* nu mai este valabilă, ele pot fi oricât de mari. Dezvoltați programul, astfel încât să poată furniza rezultate și pentru aceste cazuri. Implementați clasa *NumarMare* care să opereze în mod natural cu astfel de numere și supraîncărcați operatorii aritmetici *, +, /, %, cei de scriere/citire într-un/dintr-un flux de intrare/ieșire etc.
   Notă: O implementare a numerelor mari în C se găsește, de exemplu, în [Log06], p. 377 (pe lângă operațiile uzuale conține și extragerea radicalului de ordin dat k dintr-un număr mare).

Problema 12. Triangularizare minimală a unui poligon convex

Un poligon convex este un poligon ale cărui unghiuri sunt toate mai mici de 180° (în radiani, mai mici decât π). O triangularizare a unui poligon convex este împărțirea acestuia în triunghiuri cu ajutorul unora dintre diagonale, care nu se întretaie. Numărul de triangularizări al unui poligon convex cu $(n+2)$ vârfuri este al n-lea număr al lui Catalan: $C_n = \frac{1}{n+1} C_{2n}^n$:



Această problemă cere găsirea unei triangularizări pentru care suma perimetrelor triunghiurilor constituente să fie minimă (*triangularizare minimală*).

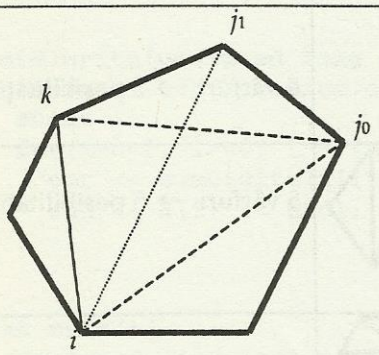
Date de intrare. În fișierul *triang.in* se găsesc mai multe puncte în plan, ce reprezintă abscisa și ordonata fiecărui vârf al poligonului, în ordinea inversă a acelor de ceasornic.

Date de ieșire. Scrieți soluția în fișierul *triang.out* ca mai jos: pe prima linie suma perimetrelor triunghiurilor formate, apoi triangularizarea găsită. Exemplu:

triang.in	triang.out
-4 -9 2 -9 8 8 -1 11 -8 8 -8 -5	Suma perimetrelor = 152.8 triangularizarea: 0 1 5 1 4 5 1 2 4 2 3 4

Analiza problemei şi proiectarea soluției

Notăm punctele date cu P_0, P_1, P_2, ..., P_{n-1}. Salvăm perimetrele tuturor triunghiurilor posibile într-un vector w (formal $w(i, j, k)$ stochează perimetrul triunghiului cu vârfurile P_i, P_j şi P_k). Suma perimetrelor triunghiurilor din care este formată o triangularizare reprezintă costul acesteia. Notăm cu $C[i][k]$, $0 \le i < k \le n-1$ costurile triangularizărilor minimale pentru subpoligoanele cu vârfurile P_i, P_{i+1}, P_{i+2}, ..., P_k. Răspunsul la problema noastră va fi deci $C[0][n-1]$.



Dacă se doreşte determinarea unei triangularizări minimale pentru subpoligonul cu vârfurile P_i, P_{i+1}, P_{i+2}, ..., P_k care conține vârful P_j, atunci aceasta se va calcula cu formula: $\min(C[i][j]) + \min(C[j][k]) + w(i, j, k)$ (această triangularizare va conține triunghiul $\Delta P_iP_jP_k$).

Costurile triangularizărilor minimale ale tuturor subpoligoanelor se vor calcula iterativ. Se începe cu subpoligoanele cu cel mai mic număr de muchii şi se sfârşeşte cu subpoligonul cu număr n de vârfuri (acesta este întregul poligon dat). Pentru subpoligonul definit de perechea de vârfuri P_i şi P_k, costul triangularizării minime este:

$$C[i][k] = \begin{cases} 0, & \text{pentru } k = i+1 \\ \min\limits_{i<j<k}\{C[i][j] + C[j][k] + w(i,j,k)\}, & \text{pentru } k > i+1 \end{cases} \quad (1)$$

Pe baza acestei formule, scriem metoda *doProcess()*, care calculează iterativ valorile $C[i][k]$ pentru toți i, k cu $0 \le i < k \le n-1$. Pentru salvarea costurilor minimale, avem nevoie doar de zona de deasupra diagonalei principale ($C[i][k]$ cu $i < k$); de aceea, vom folosi zona de dedesubtul acesteia pentru a marca drumul spre triangularizarea corespunzătoare. Metoda care va lista triunghiurile constituente va folosi recursiv această zonă.

```
ALGORITHM_COST_TRIANGULATIONS
  For (p = 1; p < n; step 1) Execute                    // dimensiune subpoligon
    For (all pairs (i, k) ← (0, p),(1, p + 1),..., (n – p – 1, n – 1)) Execute
        Calculate_with_Formel_1(C[i][k])
        C[k][i] ← j, a.î. C[i][k] = min_{i<j<k} {C[i][j] + C[j][k] + w(i, j, k)}
    End_For
  End_For
END_ ALGORITHM_COST_TRIANGULATIONS
```

Metoda recursivă care furnizează triangularizarea minimală este:

```
Write_Triangulation(i, j)
  If ( j – i > 1 ) Then
    k ← C[j][i]
    Write_Triangle(i, k, j)
    Write_Triangulation(i, k)
    Write_Triangulation(k, j)
  End_If
End_Write_Triangulation(i, j)
```

Dacă $P_1(x_1, y_1)$ şi $P_2(x_2, y_2)$ sunt două puncte în plan, atunci distanţa dintre ele este: $|P_1P_2| = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ (distanţa euclidiană). Această formulă este implementată în metoda *dist()*.

Metoda *perimeter()* calculează perimetrul unui triunghi şi are ca parametri vârfurile acestuia. Cu ajutorul celor n puncte citite, care se vor salva în vectorul vP, metoda *calculatePerimeters()* calculează perimetrele tuturor triunghiurilor posibile în ordine lexicografică a tripletelor vârfurilor: (0, 1, 2), (0, 1, 3), ..., (n – 3, n – 2, n – 1). Metoda *getW*() furnizează perimetrul unui triunghi dat. Dacă dorim determinarea perimetrului triunghiului (i_0, j_0, k_0) trebuie să găsim numărul de triplete (i, j, k) cu $0 \le i < j < k$ ce sunt mai mici lexicografic decât (i_0, j_0, k_0), adică se află înainte lui (i_0, j_0, k_0) în vectorul w. Numărul acestor triplete este suma tuturor tripletelor cu $i < i_0$, a celor pentru care $i = i_0$ şi $j < j_0$ şi a celor pentru care ($i = i_0$, $j = j_0$ şi $k < k_0$):

$$|\{(i, j, k) \mid i < i_0\}| + |\{(i, j, k) \mid i_0 = i, j < j_0\}| + |\{(i, j, k) \mid i_0 = i, j_0 = j, k < k_0\}|$$

Pentru un $i < i_0$ fixat, există C_{n-i-1}^2 triplete (numărul de perechi (j, k) cu $0 \le i < j < k$, adică toate perechile (j, k) cu $j < k$ din mulțimea $\{i_0+1, i_0+2, \ldots, n-1\}$). Pentru $i = i_0$ și un j_0 fixat există $n - j_0 - 1$ triplete (anume pentru $k = j_0+1, j_0+2, \ldots, n-1$). Pentru $i = i_0$, $j = j_0$ și $k < k_0$ există $k_0 - j_0 - 1$ triplete.

Program

```
#include <vector>
#include <fstream>
#include <cmath>

using namespace std;

typedef pair<double, double> Point;

void readVertices(ifstream &in, vector<Point*>& vP){
  double x, y;
  vP.clear();
  while(in && !in.eof()){
    if(in >> x >> y)
      vP.push_back(new Point(x, y));
  }
}

inline double sqr(double x){
  return x*x;
}

double dist(const Point P1, const Point P2){
  double x1, x2, y1, y2;
  x1 = P1.first; y1 = P1.second;
  x2 = P2.first; y2 = P2.second;
  return sqrt(sqr(x1-x2)+sqr(y1-y2));
}

double perimeter(const Point P1, const Point P2,
                 const Point P3){
  return
    dist(P1, P2) +
    dist(P2, P3) +
    dist(P3, P1);
}

void calculatePerimeters(vector<Point*>& vP,
                         vector<double> &w){
```

$$|P_1P_2| = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

```
  int i, j, k, n=vP.size();
  w.clear();
  for(i=0; i<n-2; i++)
   for(j=i+1; j<n-1; j++)
     for(k=j+1; k<n; k++){
       w.push_back(perimeter(*vP[i], *vP[j], *vP[k]));
     }
}


double getW(vector<double> &w, int i0,
            int j0, int k0, int n){
  int p=0;
  int i, j;
  for(i=0; i<i0; i++){
    p +=(n-i-1)*(n-i-2)/2;
  }
  for(j=i0+1; j<j0; j++){
    p += n-j-1;
  }
  p+=k0-j0-1;
  return w[p];
}

void doProcess(vector<double> &w,
       int n, double C[][100]){
  int i, j, k;
  double val;
  for(int p=1; p<n; p++){
    for(i=0; i<n-p; i++){
      k = i+p;
      if(1==p) C[i][k]=0;
      else{
        j=i+1;
        C[i][k] = C[i][j]+C[j][k]+getW(w, i, j, k, n);
        C[k][i]=j;
        for(j=i+2; j<k; j++){
          val = C[i][j]+C[j][k]+getW(w, i, j, k, n);
          if(val<C[i][k]) {C[i][k]=val; C[k][i]=j;}
        }
      }
    }
  }
}

void writeTriang(ofstream &out, double C[][100], int i, int j){
  if((j-i)>1){
```

Altfel, se poate „sări" peste toate tripletele mai mici lexicografic:

```
int p=0;
int i2, j2, k2;
for(i2=0; i2<i0; i2++)
  for(j2=i2+1; j2<n-1; j2++)
    for(k2=j2+1; k2<n;k2++)
      p++;
for(j2=i0+1; j2<j0; j2++)
  for(k2=j2+1; k2<n; k2++)
    p++;
p+=k0-j0-1;
return w[p];
```

```
    int k=C[j][i];
    out << "  " << i << " " << k << " " << j << endl;
    writeTriang(out, C, i, k);
    writeTriang(out, C, k, j);
  }
}

int main(){
  vector<Point*> vP;
  vector<double> w;
  int n;
  double C[100][100];
  ifstream in("triang.in");
  ofstream out("triang.out");
  readVertices(in, vP);
  n=vP.size();
  calculatePerimeters(vP, w);
  doProcess(w, n, C);
  out << " Suma perimetrelor = ";
  out.precision(4);
  out << C[0][n-1] << endl;
  out << " triangularizarea: " << endl;
  writeTriang(out, C, 0, n-1);
  return 0;
}
```

1. Citeşte vârfuri (*vP*).
2. Calculează perimetre (*vP*, *w*).
3. Construieşte C[][] pe baza *w*, *n*.
4. Scrie C[0][*n*–1].
5. Scrie triangularizare recursiv.

Probleme propuse

1. Schimbați programul pentru problema modificată, care cere determinarea unei triangularizări, pentru care suma diagonalelor care formează triangularizarea să fie minimă.
2. Scrieți un program care prezintă o simulare grafică cât mai sugestivă a determinării triangularizării minime. O modalitate de lucru ar putea fi: cu dublu-clic al mouse-ului se adaugă puncte în plan; dacă punctul curent formează cu celelalte puncte un poligon convex, atunci se va actualiza poligonul. Cu *drag-and-drop* aplicat unui vârf, poligonul se modifică şi triangularizarea minimă este întotdeauna actualizată şi subliniată cu o altă culoare şi linii îngroşate.
3. Demonstrați că formula explicită pentru *getW*(i_0, j_0, k_0) este:

$$getW(i_0, j_0, k_0) = C^3_{n-1} + C^3_{n-i_0-1} + C^2_{n-j_0-1} + C^1_{n-k_0-1}.$$

Problema 13. Înmulțirea unui șir de matrice

Pentru a înmulți două matrice A și B este necesar ca numărul de coloane ale lui A să coincidă cu numărul de linii ale lui B. Dacă matricea A are m linii și n coloane, iar B are n linii și p coloane, atunci produsul lor $A \cdot B$ are m linii și p coloane. Determinarea matricei produs se face cu $m \cdot n \cdot p$ înmulțiri elementare. A și B se pot scrie astfel ($m, n, p \in \mathbb{N}\setminus\{0\}$):

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}, \quad B = \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1p} \\ b_{21} & b_{22} & \dots & b_{2p} \\ \dots & \dots & \dots & \dots \\ b_{n1} & b_{n2} & \dots & b_{np} \end{pmatrix}$$

Atunci, matricea-produs $C = A \cdot B$ are forma:

$$C = \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1p} \\ c_{21} & c_{22} & \dots & c_{2p} \\ \dots & \dots & \dots & \dots \\ c_{m1} & c_{m2} & \dots & c_{mp} \end{pmatrix}$$

unde $c_{ij} = a_{i1} \cdot b_{1j} + a_{i2} \cdot b_{2j} + \dots + a_{in} \cdot b_{nj} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$, pentru orice $i \in \{1, \dots, m\}$, $j \in \{1, \dots, p\}$.

Pentru a determina C, este nevoie de $m \times n \times p$ înmulțiri elementare (n operații elementare pentru fiecare element c_{ij} cu $1 \le i \le m$, $1 \le j \le p$). Pentru a putea înmulți șirul de matrice $A_0A_1\dots A_n$, trebuie să fie îndeplinită condiția ca numărul de coloane ale fiecărei matrice A_i, cu i, $0 \le i < n$ să fie egal cu numărul de linii ale matricei A_{i+1}:

$$număr_coloane(A_i) = număr_linii(A_{i+1}) \text{ pentru orice } i \in \{0, \dots, n-1\}.$$

Numărul de posibilități de obținere a produsului $A_0A_1\dots A_n$ este numărul lui Catalan: $C_n = \frac{1}{n+1} C_{2n}^{n}$, egal cu numărul de parantetizări a $n + 1$ termeni:

a (b c), (a b) c	3 termeni → 2 posibilități
a (b (c d)), a ((b c) d), (a b) (c d), (a (b c)) d ((a b) c) d	4 termeni → 5 posibilități

a (b (c (d e))), a (b ((c d) e)) a ((b c) (d e)), a ((b (c d)) e) a (((b c) d) e), (a b) (c (d e)) (a b) ((c d) e), (a (b c)) (d e) (a (b (c d)))e, (a ((b c) d)) e ((a b) c) (d e), ((a b) (c d)) e ((a (b c)) d) e, (((a b) c) d) e	5 termeni → 14 posibilități

Primii termeni din şirul lui Catalan sunt: $C_0 = 1$, $C_1 = 1$, $C_2 = 2$, $C_3 = 5$, $C_4 = 14$, $C_5 = 42$, $C_6 = 132$, $C_7 = 429$, $C_8 = 1430$, $C_9 = 4862$, $C_{10} = 16796$, $C_{11} = 58786$.

Notăm cu $A(m, n)$ o matrice cu m linii şi n coloane. Dacă vom considera matricele $A_0(40, 30)$, $A_1(30, 10)$ şi $A_2(10, 2)$, atunci există două posibilități de a obține produsul lor $A_0A_1A_2$:

- $(A_0A_1)A_2$ – are nevoie în total de $40 \times 30 \times 10 + 40 \times 10 \times 2 = 12800$ produse elementare;
- $A_0(A_1A_2)$ – are nevoie în total de $40 \times 30 \times 2 + 30 \times 10 \times 2 = 3000$ produse elementare; aceasta înseamnă 23,44% din 12800. Dacă numărul de matrice creşte, atunci diferențele dintre diferite posibilități cresc considerabil.

Pentru calculul produsului a opt matrice, există $C_7 = 429$ posibilități. Dacă avem, de exemplu, matricele $A_0(34, 23)$, $A_1(23, 12)$, $A_2(12, 2)$, $A_3(2, 5)$, $A_4(5, 80)$, $A_5(80, 3)$, $A_6(3, 3)$, $A_7(3, 12)$, atunci produsul lor $A_0A_1...A_7$ poate fi astfel parantetizat:

- $(A_0(A_1((A_2A_3)A_4)))((A_5A_6)A_7)$ – are nevoie de 125800 produse elementare;
- $(A_0(A_1A_2))(((A_3(A_4A_5))A_6)A_7)$ – are nevoie de 4252 produse elementare, şi aceasta înseamnă 3,38% din numărul primei parantetizări!

Problema ce apare este deci determinarea unei parantetizări pentru un şir de matrice, care furnizează un număr minim de înmulțiri elementare.

Date de intrare. În fişierul *matrix.in* se găsesc dimensiunile unui şir de matrice, şi anume perechi (*număr_linii, număr_coloane*), câte una pe fiecare linie. Se dau maxim 100 de matrice şi numărul minim de produse elementare încape în tipul *unsigned long long*.

Date de ieşire. Scrieți pe prima linie a fişierului *matrix.out* numărul minim de produse elementare, iar pe următoarea linie o parantetizare corespunzătoare, ca în exemplu:

matrix.in	**matrix.out**
34 23 23 12 12 2	minimal = 4252 (A0(A1A2))(((A3(A4A5))A6)A7)

2 5 5 80 80 3 3 3 3 12	

Analiza problemei şi proiectarea soluției

Fie matricele A_0, A_1, ..., A_n pentru care dimensiunile matricei A_i sunt d_i şi d_{i+1} date. Introducem funcția $C(i, j)$, ce reprezintă numărul minim de produse elementare pentru calculul produsului şirului de matrice $A_i A_{i+1} \ldots A_j$. Cu această notație, obținem:

- $C(i, i) = 0$, pentru orice $i = 0, \ldots, n$ (1)
- $C(i, i+1) = d_i \cdot d_{i+1} \cdot d_{i+2}$, pentru orice $i = 0, \ldots, n-1$ (2)
- $C(i,j) = \min_{i \le k < j}(C(i,k) + C(k+1,j) + d_i \cdot d_{k+1} \cdot d_{j+1}),\ 0 \le i < j \le n$ (3)

$C(i, k)$ este numărul minim de produse elementare pentru subsecvența $A_i A_{i+1} \ldots A_k$. Rezultatul este o matrice cu d_i linii şi d_{k+1} coloane. $C(k+1, j)$ este numărul minim de produse elementare pentru subsecvența de matrice $A_{k+1} A_{k+2} \ldots A_j$. Rezultatul este o matrice cu d_{k+1} linii şi d_{j+1} coloane. Pentru a înmulți aceste două matrice rezultate, avem nevoie de încă $d_i \times d_{k+1} \cdot d_{j+1}$ produse elementare.

Vom calcula valorile iterativ pentru toate subşirurile de matrice de lungime 2, 3, ..., n. Deoarece valorile $C(i, j)$ se salvează în zona de deasupra diagonalei principale a tabloului bidimensional C[][], zona de dedesubtul acesteia ne stă la dispoziție pentru a salva valorile k ($C[j][i]$ va conține valoarea k ce conduce la minimul $C[i][j]$). Pe baza formulelor (1), (2) şi (3) scriem funcția *doProcess()*, care completează tabloul bidimensional C[][]. Dacă avem n matrice, atunci o parantetizare minimală are $C[0][n-1]$ înmulțiri elementare. Pentru a reconstrui parantetizarea minimală, scriem metoda *constructOrder()*, care completează tablourile *op*[] şi *cl*[] cu semnificațiile:

- *op*[*i*] – numărul de paranteze deschise în fața matricei A_i;
- *cl*[*i*] – numărul de paranteze închise după matricea A_i.

Pentru o pereche dată (i, j), valoarea corespunzătoare k din formula (3) este salvată în $C[j][i]$. Aceasta înseamnă că subsecvențele $A_i \ldots A_k$ şi $A_{k+1} \ldots A_j$ sunt astfel parantetizate:

```
int k=C[j][i];
if(i!=k){
 op[i]++; cl[k]++;
}
if(k+1!=j){
 op[k+1]++; cl[j]++;
}
```

şi aceste secvențe vor fi optimal parantetizate:

```
constructOrder(C,   i, k, op, cl);
constructOrder(C, k+1, j, op, cl);
```

În programul principal, se vor folosi tablourile *op*[] şi *cl*[] pentru a reconstrui întreaga parantetizare.

Program

```
#include <fstream>
#include <vector>

using namespace std;

void read(ifstream &in, ofstream& out, vector<unsigned>& d){
  int i, j;
  while(in && !in.eof()){
    if(in >> i >> j){
      if(d.size()>0 && d[d.size()-1]!=i){
        out << "Wrong input data: ";
        exit(1);
      }
      if(d.size()==0){d.push_back(i); d.push_back(j);}
      else{
        d.push_back(j);
      }
    }
  }
}

void doProcess(vector<unsigned> &d,
               unsigned long long C[][101]){

  int n = d.size()-1;
  int i, j, k, p, aux;
  for(i=0; i<n; i++) C[i][i]=0;
  for(i=0; i<n-1; i++) C[i][i+1]=d[i]*d[i+1]*d[i+2];
  for(p=2; p<n; p++){
    for(i=0; i<n-p; i++){
      j=i+p;
      C[i][j]=C[i][i]+C[i+1][j]+d[i]*d[i+1]*d[j+1];
      C[j][i]=i;
      for(k=i+1; k<j; k++){
        aux = C[i][k]+C[k+1][j]+d[i]*d[k+1]*d[j+1];
        if(C[i][j]>aux){C[i][j]=aux; C[j][i]=k;}
```

```
      }
    }
  }
}

void constructOrder(unsigned long long C[][101],
                    int i, int j, short op[100],
                    short cl[100]){
  if((j-i)>1){
    int k=C[j][i];
    if(i!=k){
      op[i]++; cl[k]++;
    }
    if(k+1!=j) {
      op[k+1]++; cl[j]++;
    }
    constructOrder(C, i, k, op, cl);
    constructOrder(C, k+1, j, op, cl);
  }
}

int main(){
  ifstream in("matrix.in");
  ofstream out("matrix.out");
  int  n, i, j;
  vector<unsigned> d;
  unsigned long long C[101][101];
  short op[100], cl[100];
  read(in, out, d);
  n=d.size()-1;
  for(i=0; i<n; i++){
    op[i]=0;
    cl[i]=0;
  }
  doProcess(d, C);
  out << "minimal = " << C[0][n-1] << endl;
  constructOrder(C, 0, n-1, op, cl);
  for(i=0; i<n; i++){
    for(j=0; j<op[i]; j++) out<<"(";
    out << "A" << i;
    for(j=0; j<cl[i]; j++) out<<")";
  }
  return 0;
}
```

Probleme propuse

1. Dezvoltați programul astfel încât să se scrie în fișierul de ieșire și parantetizarea maximală, plus procentul cu 4 zecimale exacte al reducerii celei minimale în comparație cu cea maximală, ca în exemplu:

matrix.in	matrix.out
34 23 23 12 12 2 2 5 5 80 80 3 3 3 3 12	minimal = 4252 (A0(A1A2))(((A3(A4A5))A6)A7) maximal = 125800 (A0(A1((A2A3)A4)))((A5A6)A7) 3.38 %

2. Scrieți un program C++ care analizează suma timpilor de execuție pentru diferite date de intrare. Posibilitate de implementare: mai întâi, clasa *CMatrix*, ce reprezintă conceptul de matrice din matematică cu o serie de operatori (printre care citire, scriere, înmulțire, adunare etc.); se vor genera aleatoriu șiruri de matrice cu condiția din enunț și pentru fiecare se va calcula parantetizarea minimală și maximală. Aceste parantetizări, plus cea inițială se vor utiliza pentru a calcula produsul lor și, pentru fiecare, timpii de calcul se vor aduna unor variabile corespunzătoare. Raportul se va afișa într-un fișier *report.txt*: pentru fiecare caz șirul de matrice, apoi parantetizările maximală și minimală, produsul lor obținut cu fiecare dintre ele, plus ordinea inițială (trebuie să fie aceleași), timpii de execuție pentru determinarea fiecărui produs. La sfârșit, se vor afișa timpii de execuție globali, adică sumele pentru parantetizările minimală, maximală, inițială pentru toate cazurile date.
   Notă. Un program C care calculează produsul a două matrice se găsește, de exemplu, în [Log06], p. 145.

Problema 14. *Edit-Distance*

Metrica în matematică este o noțiune abstractă și nu se referă neapărat la distanța în spațiu (geometrică), care este o metrică particulară. O funcție $d: M \times M \to \mathbb{R}$ reprezintă o *metrică* atunci când sunt îndeplinite următoarele condiții:

- $d(x, y) \geq 0$ pentru orice $x, y \in M$, $d(x, y) = 0 \Leftrightarrow x = y$;
- simetria: $d(x, y) = d(y, x)$ pentru orice $x, y \in M$;
- egalitatea triunghiului: $d(x, y) \leq d(x, z) + d(z, y)$ pentru orice $x, y, z \in M$.

O altă metrică cunoscută este *distanța Hamming*. Ea se definește ca fiind numărul de poziții în care două cuvinte de aceeași lungime nu coincid, atunci când se compară caracterele de pe aceste poziții. Exemplu: *distanța Hamming* pentru cuvintele *abcd* și *amcn* este 2 pentru că pe două poziții (2 și 4) caracterele nu coincid. Nu este greu să se demonstreze că *distanța Hamming* este o metrică.

O altă metrică este *Edit-Distance* (distanța de editare), numită și *Levenshtein-Distance* (după omul de știință rus Vladimir Levenshtein [n. 1935], care a prezentat algoritmul în 1965). Această distanță este definită pentru cuvinte formate pe un alfabet dat. Asupra unui cuvânt se pot folosi trei feluri de transformări: ștergerea sau introducerea unui caracter, înlocuirea unui caracter cu un altul. *Edit-Distance* pentru două cuvinte W_1 și W_2 este definită ca fiind numărul minim de transformări necesare pentru a obține W_2 din W_1. Exemplu: *d(anne, marie)* = 3, deoarece cuvântul *anne* se transformă în minim trei pași în *marie*: *anne* → *manne* → *marne* → *marie* (*inserare*: poziția 1; *înlocuire*: pozițiile 3 și 4).

Edit-Distance satisface cele trei proprietăți ale unei metrici. Ea se folosește în practică printre altele la: prelucrarea și verificarea corectitudinii textelor, analiza ADN-ului, recunoașterea plagiatelor, *File Revision, Remote Screen Update Problem.*

În această problemă, se cere determinarea distanței de editare dintre două cuvinte date. Cuvintele sunt formate din literele mici de la *a* la *z* și au lungimea maximă 100.

Date de intrare. În fișierul *edit.in* se găsesc mai multe perechi de cuvinte, câte una pe linie.

Date de ieșire. Scrieți în fișierul *edit.out* distanța de editare pentru fiecare astfel de pereche (*D* – ștergere, *I* – inserare, *T* – transformare). Exemplu:

edit.in	edit.out
anne marie klipp klar iute dulce graba treaba	d(anne, marie) = 3 anne I(1) --> manne T(3) --> marne T(4) --> marie ========== d(klipp, klar) = 3 klipp D(3) --> klpp T(3) --> klap T(4) --> klar ========== d(iute, dulce) = 3 iute T(1) --> dute I(3) --> dulte T(4) --> dulce ========== d(graba, treaba) = 2 graba T(1) --> traba I(3) --> treaba ==========

Analiza problemei şi proiectarea soluției

Fie cuvintele $x = x_1x_2...x_m$ şi $y = y_1y_2...y_n$ de lungimi m, respectiv n. Dacă nu există litere comune în cele două cuvinte, atunci distanța de editare este $max(m, n)$. Această distanță este mai mică atunci când există litere comune. O modalitate de rezolvare este construirea unui tablou bidimensional $T[][]$ cu $n+1$ linii şi $m+1$ coloane cu semnificația că $T[i][j]$ este *Edit-Distance* dintre cuvintele $x_1x_2...x_j$ $(0 \le j \le m)$ şi $y_1y_2...y_i$ $(0 \le i \le n)$. La început se inițializează linia 0 cu valorile 0, 1, 2, ..., m (pentru a transforma cuvântul $x_1x_2...x_j$ în cuvântul vid, avem nevoie de j operații de ştergere). Coloana 0 se va inițializa cu valorile 0, 1, 2, ..., n (pentru a transforma cuvântul vid în cuvântul $y_1y_2...y_i$, avem nevoie de i operații de inserție). Pentru a obține distanța $T[i][j]$ între cuvintele $x_1x_2...x_j$ $(1 \le j)$ şi $y_1y_2...y_i$ $(1 \le i)$, există trei posibilități:

- din $T[i-1][j-1]$: în cazul $x_j = y_i$, nu este nevoie de nici o transformare şi putem avea $T[i][j] = T[i-1][j-1]$; dacă $x_j \ne y_i$, atunci x_j se înlocuieşte cu y_i şi putem avea $T[i][j] = T[i-1][j-1] + 1$;
- din $T[i-1][j]$: $y_1y_2...y_{i-1}$ este obținut din $x_1x_2...x_j$ cu $T[i-1][j]$ operații, adică $y_1y_2...y_i$ se obține din $x_1x_2...x_j$ prin $T[i-1][j] + 1$ operații, atunci când ultima dintre ele este inserarea literei y_i;
- din $T[i][j-1]$: numărul minim de paşi pentru a obține $y_1y_2...y_i$ din $x_1x_2...x_{j-1}$ este $T[i][j-1]$, adică $y_1y_2...y_i$ se obține din $x_1x_2...x_j$ prin $T[i][j-1] + 1$ operații, dacă ultima dintre ele este ştergerea literei x_j.

Notăm $b(i, j) = 0$, dacă $x_j = y_i$ şi $b(i, j) = 1$, dacă $x_j \ne y_i$. Atunci:

$$T[i][j] = \begin{cases} j, & \text{dacă } i = 0, j = 1,...,m \\ i, & \text{dacă } j = 0, i = 1,...,n \\ \min(T[i-1][j-1] + b(i, j),\ T[i-1][j] + 1,\ T[i][j-1] + 1), & \text{dacă } i = 1,...,n,\ j = 1,...,m \end{cases}$$

Distanța de editare dintre $x_1x_2...x_m$ şi $y_1y_2...y_n$ este atunci $T[n][m]$. Pentru primele două exemple, tabloul bidimensional $T[][]$ arată astfel:

	_	a	n	n	e
_	0	1	2	3	4
m	1	1	2	3	4
a	2	1	2	3	4
r	3	2	2	3	4
i	4	3	3	3	4
e	5	4	4	4	3

	_	k	l	i	p	p
_	0	1	2	3	4	5
k	1	0	1	2	3	4
l	2	1	0	1	2	3
a	3	2	1	1	2	3
r	4	3	2	2	2	3

Exemple Edit-Distance

Metoda *getEditDistance()* construieşte tabloul *T*[][]. Valorile *T*[*i*][*j*] vor fi iterativ calculate, linie cu linie, de la stânga spre dreapta, folosind formula de mai sus. Pentru a găsi apoi şirul de transformări, pornim cu *T*[*n*][*m*] şi ne deplasăm în sens invers până la câmpul *T*[0][0]. La fiecare pas *T*[*i*][*j*] decidem care dintre cele trei operaţii a condus la obţinerea valorii respective şi stabilim una dintre ele. *T*[*i*][*j*] a fost obţinut pe baza uneia dintre celulele vecine: de pe diagonală deasupra-stânga, de deasupra sau din stânga. Vom folosi pentru fiecare dintre ele operaţia inversă pe cuvântul actual. O ştergere devine inserţie şi invers, înlocuirea se face în sens invers.

Metoda recursivă *writeSolution()* realizează aceste operaţii. Ea se va apela cu cuvântul-scop ca parametru şi determină o procedură de obţinere a sa din cuvântul iniţial pe baza tabloului *T*[][]. Metoda modifică cuvântul actual, se apelează recursiv cu noul parametru şi scrie după aceea operaţia executată şi cuvântul curent în fişierul de ieşire. Pot să existe mai multe drumuri optime, noi vom reconstrui însă doar unul dintre ele.

Dacă de exemplu $x_m = y_n$ şi $T[n-1][m-1] = T[n][m]$, atunci $T[n][m]$ este obţinut din $T[n-1][m-1]$ fără aplicarea vreunei operaţii, caz în care setăm variabila *flag* pe *true* (am găsit un predecesor pentru $T[n][m]$!) şi apelăm metoda *writeSolution()* cu noii parametri:

```
flag = true;
writeSolution(m-1, n-1, T, x, y, yaux, out);
```

Dacă $x_m \neq y_n$ şi $T[n-1][m-1] + 1 = T[n][m]$, atunci $T[n][m]$ se obţine din $T[n-1][m-1]$ prin înlocuirea unui caracter, caz în care setăm *flag* pe *true* (am găsit un predecesor pentru $T[n][m]$!). Aplicăm pe cuvântul curent *yaux* operaţia inversă, apelăm metoda *writeSolution()* cu noii parametri şi scriem operaţia executată în fişierul de ieşire:

```
yaux.replace(n-1, 1, x.substr(m-1, 1));
flag = true;
writeSolution(m-1, n-1, T, x, y, yaux, out);
out << yaux << " T(" << n << ") --> ";
```

Acelaşi procedeu se foloseşte şi pentru cazul când predecesorul lui $T[n][m]$ este $T[n][m-1]$ ($T[n][m] = T[n][m-1] + 1$) sau predecesorul lui $T[n-1][m]$ este $T[n][m]$ ($T[n][m] = T[n-1][m] + 1$): transformare inversă, apel *writeSolution()* cu noi parametri, scrierea operaţiei inverse. Remarcăm că în C++ literele şirului încep cu indexul 0; de aceea, testăm, de exemplu, *if*($x[m-1]==y[n-1]$..., în loc de $x[m] == y[n]$. În aceeaşi manieră sunt adaptate şi funcţiile pentru şiruri (*erase, insert, replace, ...*).

Program

```
#include <string>
#include <fstream>

using namespace std;

short getEditDistance(string x, string y, short T[][101]){
  short m, n, i, j, res;
  m=x.length(); n=y.length();
  if(!m) res = n;
  if(!n) res = m;
  if(m && n){
    for(i=0; i<=m; i++) T[0][i]=i;
    for(i=0; i<=n; i++) T[i][0]=i;
    for(i=1; i<=n; i++)
      for(j=1; j<=m; j++){
        if(x[j-1]==y[i-1]){
          T[i][j]=T[i-1][j-1];
        }
        else{
          T[i][j]=T[i-1][j-1]+1;
        }
        if(T[i][j]>T[i-1][j]+1) T[i][j]=T[i-1][j]+1;
        if(T[i][j]>T[i][j-1]+1) T[i][j]=T[i][j-1]+1;
       }
      res = T[n][m];
    }
  return res;
}

void writeSolution(int m, int n, short T[][101],
                   string& x,  string& y, string yaux,
                   ofstream &out){
  bool flag = false;
  if(m || n){
    if(m>0 && n>0){
      if(x[m-1]==y[n-1]){
        if(T[n-1][m-1]==T[n][m]){
          flag = true;
          writeSolution(m-1, n-1, T, x, y, yaux, out);
        }
      } else{
        if(T[n-1][m-1]+1==T[n][m]){
          yaux.replace(n-1, 1, x.substr(m-1, 1));
          flag = true;
          writeSolution(m-1, n-1, T, x, y, yaux, out);
          out << yaux << " T(" << n << ") --> ";
```

```
        }
      }
    };

    if(n>0 && !flag){
      if(T[n][m]==T[n-1][m]+1){
        flag=true;
        yaux.erase(n-1, 1);
        writeSolution(m, n-1, T, x, y, yaux, out);
        out << yaux << " I(" << n << ") --> " ;
      }
    };

    if(m>0 && !flag){
      if(T[n][m]==T[n][m-1]+1){
        yaux.insert(n, x.substr(m-1, 1));
        writeSolution(m-1, n, T, x, y, yaux, out);
        out <<  yaux << " D(" << n+1 << ") --> ";
      }
    };
  };
}


int main(){

  string x, y;
  short T[101][101], res;
  ifstream in("edit.in");
  ofstream out("edit.out");
  while(in && !in.eof()){
    if(in >> x >> y){
      res = getEditDistance(x, y, T);
      string yaux = y.substr(0, y.length());
      out << endl;
      out << "d(" << x << ", " << y << ") = "
          << res << endl;
      writeSolution(x.length(), y.length(), T,
                    x, y, yaux, out);
      out << y << endl << "========================";
    }
  }

  return 0;

}
```

Probleme propuse

1. Modificați metoda *writeSolution()* astfel încât să se afișeze toate modalitățile de a obține distanța de editare minimă.
2. Presupunem că pentru cele trei operații sunt asociate costuri diferite: pentru înlocuire C_t, pentru ștergere C_d, pentru inserarea unei litere C_i. Aceste costuri sunt date în fișierul de intrare ca triplete (C_t, C_d și C_i) imediat după perechile de cuvinte. Dezvoltați programul astfel încât să se afișeze o modalitate de transformare ce are cost minim. Exemplu:

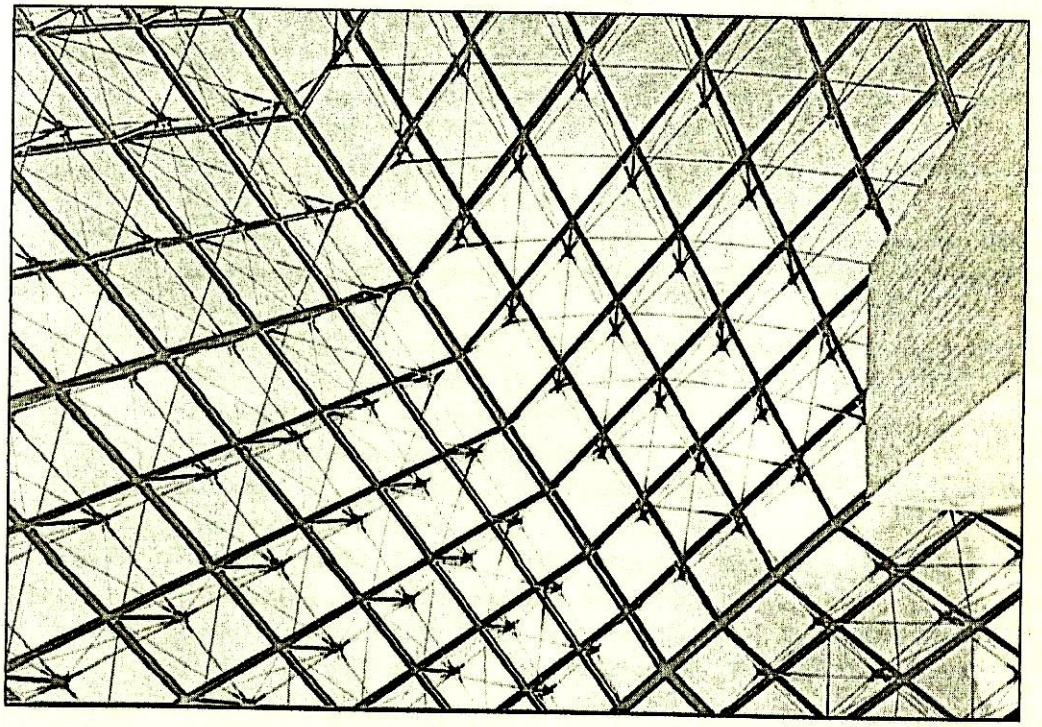
edit1.in	edit1.out
probleme loesungen 4 2 1 baerchen zwerglein 5 3 1 boese gute 2 3 1	d(probleme, loesungen) = 17 probleme D(1) --> robleme D(1) --> obleme D(1) --> bleme D(1) --> leme I(2) --> loeme D(4) --> loee I(4) --> loese I(5) --> loesue I(6) --> loesune I(7) --> loesunge I(9) --> loesungen ========== d(baerchen, zwerglein) = 19 baerchen D(1) --> aerchen D(1) --> erchen I(1) --> zerchen I(2) --> zwerchen D(5) --> zwerhen D(5) --> zweren I(5) --> zwergen I(6) --> zwerglen I(8) --> zwerglein ========== d(boese, gute) = 7 boese D(1) --> oese T(1) --> gese T(2) --> guse T(3) --> gute ==========

Modificați programul astfel încât să afișeze toate modalitățile de a obține acest cost minim, pentru cazul că există mai multe.

3. Notăm cuvântul vid cu λ. Putem folosi și un algoritm de tip *Divide et Impera* bazat pe următoarele formule:
   - $d(\lambda,\lambda) = 0$;
   - $d(x_1x_2\ldots x_i, \lambda) = i$ pentru orice $i \geq 1$;
   - $d(\lambda, y_1y_2\ldots y_j) = j$ pentru orice $j \geq 1$.

$$d(x_1x_2\ldots x_m, y_1y_2\ldots y_n) = \begin{cases} \max\{m,n\}, \text{dacă nu există litere comune} \\ \min\limits_{x_i=y_j}(d(x_1\ldots x_{i-1}, y_1\ldots y_{j-1}) + d(x_{i+1}\ldots x_m, y_{j+1}\ldots y_n)), \text{dacă există cel} \\ \text{puțin o literă comună} \end{cases}$$

Scrieți un program care se bazează pe aceste formule.



Piramida de sticlă de la Luvru, Paris